LE SYSTÈME SYNTAX™

MANUEL D'UTILISATION ET DE MISE EN ŒUVRE SOUS UNIX™

Pierre Boullier Philippe Deschamp
INRIA–Rocquencourt
BP 105
78153 Le Chesnay Cedex
France

Mise à jour de Août 1991,
édition du 18 octobre 2007.

Ce document permet l'utilisation du système SYNTAX[1] de production de traducteurs, implanté sous UNIX[2] ; il décrit la version de distribution 3.5 de juillet 1988. Tous commentaires, remarques, questions et suggestions seront bienvenus, qu'ils concernent le système lui-même ou le présent manuel : s'adresser à l'Équipe Langages et Traducteurs de l'INRIA par courrier, ou, par téléphone, au +33(1)39-63-56-05, par courrier électronique à syntax@minos.inria.fr (liste de personnes) ou directement aux auteurs[3].

[1] SYNTAX est une marque déposée de l'INRIA.
[2] UNIX est une marque déposée des Laboratoires Cloche (AT&T Bell Laboratories).
[3] {boullier,deschamp}@minos.inria.fr

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Le système SYNTAX regroupe un ensemble d'outils dont le but premier est de faciliter la conception et la réalisation de *traducteurs*—principalement, mais non exclusivement, dans le domaine de la compilation. Ces outils permettent d'une part la *construction* d'analyseurs (syntaxiques, lexicographiques et sémantiques), d'autre part la *compilation* de textes sources à l'aide des analyseurs créés au préalable.

Les buts poursuivis sont donc du même ordre que ceux qui ont présidé à la définition des utilitaires LEX et YACC de UNIX, mais il est beaucoup plus puissant et performant, en particulier en ce qui concerne le traitement des erreurs. De plus, les analyseurs produits peuvent être associés à des formes de traitement sémantique de plus haut niveau que celle disponible dans YACC, c'est-à-dire la simple exécution de fragments de code associés à chaque production.

SYNTAX comprend principalement les modules suivants :

- BNF : mise sous forme interne des définitions syntaxiques ;
- CSYNT : constructeur syntaxique ;
- LECL : constructeur lexicographique ;
- RECOR : traitement des erreurs ;
- TABLES_C : production des tables d'analyse en langage C ;
- les outils réalisant l'analyse des textes source : `sxscanner`, `sxparser`, ..., ou aidant à cette analyse : `sxsrc_mngr`, `sxstr_mngr`, *et cetera*.

La majeure partie de ce manuel est consacrée à la description de ces modules, tant en ce qui concerne leurs fonctionnalités qu'en ce qui concerne leur mise en œuvre sous le système UNIX. En outre, le chapitre 5 est consacré aux divers moyens de réaliser des traitements *sémantiques* à l'aide de SYNTAX.

1.1 Les processeurs de base

1.1.1 Introducteur syntaxique : BNF

BNF est le processeur de base pour la mise sous forme interne des définitions syntaxiques ; il lit une grammaire "indépendante du contexte", effectue quelques vérifications simples de cohérence, et produit une forme interne de cette grammaire (des "tables") qui sera utilisée par les autres processeurs.

La grammaire d'entrée est écrite dans un langage proche de la "Backus-Naur Form" bien connue. Les non-terminaux et les terminaux sont distingués

lexicalement. Chaque alternative donne lieu à une production différente.

BNF accepte des grammaires ambigües, à condition que ces ambigüités puissent être levées par la donnée de niveaux de priorité (voir la section concernant CSYNT et PRIO ci-dessous). De plus, l'analyse syntaxique peut être influencée par des *prédicats* et des *actions* programmés par l'auteur de la grammaire, ce qui permet de traiter des langages non-déterministes, voire dépendants du contexte.

1.1.2 Constructeur syntaxique : CSYNT et PRIO

CSYNT est le constructeur syntaxique. Il lit les tables produites par BNF et construit un analyseur syntaxique ascendant en utilisant la méthode LALR(1). Les conflits détectés lors de la construction de l'analyseur qui ne seraient pas résolus par la lecture d'une unité lexicale en avance peuvent l'être de plusieurs autres manières :

- par l'auteur de la description, soit en utilisant des prédicats et des actions comme décrit dans la section consacrée à BNF, soit en forçant des niveaux de priorité (voir ci-dessous) ;
- automatiquement par CSYNT, grâce à l'utilisation par celui-ci de règles de résolution prédéfinies (par exemple *Shift* prend précédence sur *Reduce*).

La spécification de résolution des conflits est traitée par PRIO. Elle se fait en utilisant une syntaxe et une sémantique très proches de celles de YACC, mais la description est séparée de la grammaire proprement dite.

Sur option, CSYNT produit un listage détaillé décrivant les conflits détectés, ce qui permet à un utilisateur averti de les comprendre facilement.

Le second composant de CSYNT réduit très fortement (usuellement de plus de 95%) la taille des tables représentant l'automate implantant l'analyseur syntaxique. En outre, sur option, il est capable d'éliminer *totalement* les réductions par des productions simples, ce qui augmente en général la vitesse de l'analyse.

1.1.3 Constructeur lexical : LECL

LECL est le constructeur lexical. Il lit les tables produites par BNF (en particulier la table donnant pour chaque terminal ceux qui peuvent le suivre), ainsi qu'une description lexicale du langage, et produit des tables décrivant l'analyseur lexical engendré.

Certains terminaux de la grammaire (tels ceux qui représentent les identificateurs du langage ou ses constantes), ainsi que les *commentaires* du langage, doivent être décrit par une *expression régulière* sur un alphabet dont les lettres sont des classes de caractères. LECL propose un certain nombre d'opérations ensemblistes pour définir ces classes. Les opérations disponibles pour la construction d'expressions régulières sont la séquence, l'alternative, la répétition, l'optionnalité et le groupement. Il est aussi possible de définir des *abréviations* correspondant à des "morceaux" d'expressions régulières, et de les utiliser pour définir d'autres expressions.

Pour construire l'automate d'états finis implantant l'analyseur lexical, LECL utilise des techniques qui sont dérivées de la méthode LR pour la construction d'analyseurs syntaxiques ; ceci lui permet d'obtenir directement un automate déterministe.

Les (inévitables) conflits détectés lors de la construction de l'analyseur qui ne seraient pas résolus par la lecture d'un caractère en avance peuvent l'être de

plusieurs autres manières :

- par l'auteur de la description, soit en réduisant les contextes déterminés à partir de la grammaire, soit en utilisant des prédicats et des actions semblables à ceux du niveau syntaxique, soit en forçant des niveaux de priorité ;
- automatiquement par LECL, soit statiquement en utilisant des règles de résolution prédéfinies (par exemple *Shift* prend précédence sur *Reduce*), soit dynamiquement en lisant un nombre éventuellement *non borné* de caractères en avance.

LECL construit aussi un automate très performant pour la reconnaissance des mots clés.

Les automates produits par LECL sont représentés par des tables. Sur option, l'analyseur lexical peut aussi être produit sous la forme d'un programme C spécifique, ce qui permet d'augmenter la vitesse d'analyse.

1.1.4 Traitement des erreurs : RECOR

L'avantage le plus appréciable de SYNTAX sur YACC est son puissant traitement d'erreurs ; il faut d'ailleurs reconnaître que YACC est particulièrement rustique sur ce point.

Tout traitement d'erreur se décompose en trois ou quatre phases :

1. détection,
2. affichage,
3. tentative de correction,
4. rattrapage, si la correction a échoué.

En ce qui concerne la détection d'erreur, les méthodes d'analyse employées par SYNTAX possèdent la propriété du préfixe correct, ce qui garantit qu'une erreur est détectée dès que la partie du texte déjà lue ne peut pas faire partie d'un texte correct.

En ce qui concerne l'affichage, SYNTAX émet des messages d'erreur très clairs, avec rappel au terminal de la ligne du texte source et marqueur sous l'erreur. De plus, si les analyseurs sont construits en conséquence, un listage est produit, contenant le texte source et les messages d'erreur au "bon" endroit.

L'analyse se poursuit après une correction ou un rattrapage.

1.1.4.1 Correction locale

Quand une erreur est détectée, l'analyseur produit virtuellement toutes les parties de texte syntaxiquement correctes à partir du point en erreur et les compare à une liste ordonnée de *modèles de correction*. Ces modèles, fournis à la construction par l'auteur de la grammaire, peuvent spécifier un nombre *quelconque* de suppressions, d'insertions *vel* de remplacements dans le texte source. Si l'une des parties de texte engendrées correspond à l'un de ces modèles, la correction est acceptée : le nouveau texte vient remplacer l'ancien, et l'analyse reprend.

De nombreux dispositifs annexes permettent à l'auteur de la spécification de contrôler très finement le mécanisme de correction : correction de fautes d'orthographe sur les mots clés, longueur de validation, terminaux clés, ensembles

Don't Delete et *Don't Insert*, possibilité d'agir sur le terminal précédant l'erreur, ...

Notons que l'analyseur lexical bénéficie du même mécanisme de correction que l'analyseur syntaxique. Notons aussi que ce mécanisme est totalement indépendant de la grammaire et ne nécessite pas de modifier cette dernière.

1.1.4.2 Rattrapage global

Si la correction locale échoue—ce qui se produit dans moins de 20% des cas en pratique—, l'analyseur doit tout de même pouvoir analyser le reste du texte. Ceci est possible grâce au rattrapage global.

Pour ce qui est du rattrapage syntaxique, le texte est lu sans analyse jusqu'à rencontrer un *terminal clé* tel qu'il existe dans la pile un état ayant une transition non-terminale valide pouvant être suivie du dit terminal. L'analyseur est ensuite redémarré avec ce terminal en entrée, après que la pile ait été écrêtée jusqu'à cet état. L'effet net de ce mécanisme est d'ignorer une partie de texte entourant le point d'erreur, ce qui est un pis-aller mais permet d'analyser la suite du texte.

Le rattrapage lexical consiste à détruire purement et simplement le caractère en erreur.

L'action combinée des deux phases du traitement d'erreur de SYNTAX est en pratique très satisfaisante, et en tous cas incomparablement supérieure à celle de YACC.

1.1.4.3 Spécification du traitement d'erreur

Tous les aspects du traitement d'erreur sont décrits séparément de la grammaire, qu'il s'agisse de l'affichage, de la correction locale ou du rattrapage global, et ce aussi bien pour l'analyse lexicale que pour l'analyse syntaxique. L'auteur de cette description spécifie en particulier les modèles de correction, les terminaux clés et les messages d'erreur : aucune partie de message n'étant fabriquée directement par SYNTAX, ceci permet de les adapter par exemple à la langue naturelle des utilisateurs ou à leur niveau de compétence.

Cette description est traitée par le processeur RECOR.

1.1.5 Production des tables : TABLES_C

Ce dernier processeur lit les tables produites par les autres constructeurs et construit un "programme" C, composé essentiellement de structures de données initialisées, contenant toutes les informations nécessaires au système d'exécution de SYNTAX, décrit ci-dessous, pour traiter le langage en question. Ce programme doit être compilé et lié au système d'exécution pour former un analyseur complet.

1.2 Le Système d'Exécution

SYNTAX ne produit pas directement un programme exécutable, mais un ensemble de données C à compiler et lier avec différents modules du système d'exécution. Ces modules comprennent bien entendu un analyseur lexical et un analyseur syntaxique qui vont interpréter les tables du langage à analyser, mais

aussi de nombreux modules utilitaires permettant de réaliser à peu de frais un "squelette" de compilateur :

- gestionnaire du texte source, permettant en particulier de traiter les *includes* ;
- gestionnaire des messages d'erreur, permettant de les afficher au cours de l'analyse et de les stocker pour les inclure dans le listage ;
- gestionnaire de chaînes de caractères, permettant de stocker des textes de longueur quelconque, de leur affecter un "numéro unique" et d'y accéder facilement ;
- enchaîneur de passes ;
- modules de traitement d'erreur ;
- module de production du listage ;
- gestionnaire de chaînes de bits destinées à représenter des ensembles...

Tous ces modules sont disponibles sous forme binaire dans une bibliothèque manipulable par l'éditeur de liens, et aussi sous forme source, ce qui permet de les modifier pour les adapter à des besoins particuliers.

1.3 Le Traitement Sémantique

L'utilité d'un pur analyseur lexico-syntaxique est faible. Certes, il peut être utile de savoir si un programme est syntaxiquement correct avant de le soumettre à un compilateur, *a priori* plus lent, mais, dans la plupart des cas, il faut compléter l'analyse par un traitement sémantique. SYNTAX propose plusieurs méthodes pour écrire un tel traitement, décrites ci-après. Pour chacune d'elles, un unique fichier source contient à la fois la grammaire du niveau syntaxique et la spécification du traitement sémantique. Cette spécification se fait en insérant du texte après chaque production de la grammaire. Un unique processeur traite à la fois la grammaire, avec des fonctionnalités héritées de BNF, et la spécification de la sémantique.

1.3.1 Actions

La méthode de plus bas niveau, mais aussi la plus puissante, est d'associer un numéro d'action à chaque production, et d'écrire une procédure comportant un fragment de programme pour chacun de ces numéros. Lors de chaque réduction, l'analyseur appellera cette procédure en lui passant le numéro de l'action associée à la production par laquelle on réduit. Cette action pourra alors accéder à la pile d'analyse et effectuer un traitement approprié.

Cette forme de sémantique est la plus délicate à utiliser et la plus fastidieuse à manipuler, mais c'est aussi la plus puissante puisque tout y est permis. On pourrait la considérer comme "le langage d'assemblage" du traitement sémantique. Cette comparaison est d'autant plus justifiée que, pour l'analyseur syntaxique, tout traitement sémantique est constitué d'actions appelées à chaque réduction...

Cette forme de sémantique est traitée par le processeur SEMACT.

1.3.2 Actions à la YACC

Cette forme de traitement sémantique est un peu plus structurée que la précédente en ce sens que les fragments de code implantant les actions sont écrits directement après les productions, sous forme de blocs C (entre accolades). Le processeur YAX produit lui-même les numéros d'action et la procédure correspondante.

En outre, les accès à la pile sont "déguisés" par l'emploi d'une notation héritée de YACC, dont YAX est fortement inspiré. La gestion de la pile est aussi effectuée automatiquement. Il faut noter tout de même une différence importante : YAX n'accepte pas, contrairement à YACC, que de tels fragments de code soient insérés *au milieu* des parties droites des productions.

SYNTAX propose un traducteur, nommé YSX, transformant (purement syntaxiquement) une spécification pour YACC en une spécification pour YAX. Les actions au milieu des productions sont correctement traduites en les attachant à des non-terminaux dérivant la chaîne vide, produits automatiquement.

Une version de YAX avec des actions en Pascal est actuellement à l'étude.

1.3.3 Attributs sémantiques purement synthétisés

La formalisation des notions d'*accès à la pile* et de *gestion de la pile* employées ci-dessus est la notion de *grammaire attribuée*. La description de cette technique demande trop de place pour figurer ici, mais disons seulement qu'elle permet d'exprimer de façon purement *déclarative* tous les calculs dirigés par la syntaxe.

SYNTAX propose une forme restreinte de grammaires attribuées, dans laquelle les valeurs des attributs sont calculées de bas en haut de l'arbre de dérivation (purement synthétisées). Le calcul de ces valeurs peut être décrit soit en C (processeur TABC), soit en PASCAL (processeur TABACT). Dans ce dernier cas, plusieurs modules d'interface, inclus dans le système d'exécution, permettent d'accéder en PASCAL aux fonctionnalités de la version C.

1.3.4 Arbres abstraits

Si le traitement sémantique désiré ne peut pas s'effectuer en une passe au cours de l'analyse syntaxique, il faut construire un arbre représentant le texte source et susceptible d'être parcouru plusieurs fois. SYNTAX propose une telle forme de traitement sémantique, dans laquelle les arbres produits sont des arbres abstraits, ce qui signifie qu'ils ne contiennent pas d'information purement syntaxique comme les terminaux non génériques et les productions simples. De plus, les notions du langage exprimant des listes sont effectivement représentées par des nœuds listes, et non pas par des peignes.

La spécification de ce traitement se fait en attachant des noms de nœud à certaines productions. La description est traitée par le processeur SEMAT. Ce dernier produit, outre des tables permettant à un module du système d'exécution de construire l'arbre au cours de l'analyse syntaxique, des "squelettes" de programmes C, dans lesquels les actions à effectuer sur chaque nœud visité peuvent être insérées. Un autre module du système d'exécution permet de parcourir ces arbres du haut en bas et de gauche à droite—on peut aussi "programmer" des parcours différents—en appelant les actions associées à chaque nœud.

FIG. 1.1 – Exemple de spécification de paragraphage.

Voici un extrait d'une grammaire de PASCAL "bien présentée" :

```
<PROC DECL>      = <PROC HEAD>
                   <BLOCK> ;
<PROC HEAD>      = procedure  %ID  <FORMALS *>  ";" ;
<BLOCK>          = <DECL *>
                   <COMPOUND STMT>  ";" ;
<COMPOUND STMT> = begin
                       <STMT +>
                   end ;
```

PARADIS produit automatiquement un paragrapheur, à partir de la spécification précédente ; ce paragrapheur est capable de transformer le texte suivant :

```
ProCedure TRuc
; begin instruction1;
Instruction2           end
;
```

en la forme paragraphée suivante :

```
procedure TRUC ;
begin
    INSTRUCTION1 ;
    INSTRUCTION2
end ;
```

Des informations peuvent être attachées à chaque nœud. Un autre module du système d'exécution produit une représentation "semi-graphique" (alphanumérique) de ces arbres. Le processeur MIX permet d'automatiser la mise à jour des "passes sémantiques" après modification de la grammaire *vel* de la spécification des arbres abstraits.

1.3.5 Paragraphage

L'expérience prouve rapidement qu'il est beaucoup plus agréable et efficace de travailler sur des programmes dont la mise en page reflète la structure. Des processeurs permettant de produire automatiquement une version *paragraphée* de textes écrits sans prendre en compte leur structure sont donc bienvenus.

SYNTAX propose un moyen de construire automatiquement de tels paragrapheurs, en utilisant une technique basée sur la remarque suivante : quand on écrit la grammaire d'un langage, on présente souvent les productions de cette grammaire de la même façon qu'on voudrait voir paragrapher les textes de ce langage (voir la figure 1.1).

Le processeur PARADIS permet donc d'obtenir à peu de frais un paragrapheur pour un langage une fois qu'on en a écrit la grammaire, en présentant cette dernière de façon à ce qu'elle soit agréable à l'œil.

En fait, cette approche est un peu simpliste. PARADIS propose donc, en plus de ce principe général, des directives de paragraphage qui permettent d'obtenir

des mises en page plus sophistiquées. En outre, les problèmes classiques de longueur bornée de la ligne de sortie et de placement des commentaires sont traités de manière généralement satisfaisante (étant entendu qu'ils n'ont pas de solution complète).

1.3.6 Utilitaires divers

Outre les constructeurs de base et les outils traitant la sémantique, SYNTAX comprend un certain nombre d'utilitaires :

- CX, un générateur de références croisées pour le langage C ;
- PPC et PPADA, des paragrapheurs pour les langages C et Ada[1] respectivement.

Ces outils ont bien sûr été construits à l'aide de SYNTAX.

1.4 Mise en Œuvre de SYNTAX sur UNIX

SYNTAX est actuellement disponible sur de nombreuses machines munies du système d'exploitation UNIX (ou l'un de ses dérivés). Soit `$sx` le répertoire où a été installé SYNTAX. Le contenu des différents sous-répertoires de `$sx` est le suivant :

`$sx/bin` contient les versions binaires exécutables de tous les processeurs de SYNTAX. Habituellement, ces fichiers sont aussi liés (au sens UNIX) dans un répertoire "standard" comme `/usr/local/bin`, ce qui évite aux utilisateurs d'avoir à changer leur "`PATH`".

`$sx/doc` contient la documentation. Outre un fichier de nom `NOUVEAUTES`, qui résume l'évolution des diverses versions distribuées, il contient les sous-répertoires `man` (les "pages de manuel", habituellement liées ou copiées dans `/usr/man` ou autre pour accès immédiat) et `help` (courtes descriptions des commandes de SYNTAX, utilisables en particulier par la fonction "*help*" de `tcsh(1)`), ainsi que divers fichiers facilitant la mise en œuvre de SYNTAX.

`$sx/incl` contient des fichiers "*include*" permettant l'accès aux modules du système d'exécution. Le plus important est `sxunix.h` qui est utilisable en C. Il existe aussi des versions PASCAL, disponibles dans des sous-répertoires.

`$sx/lib` contient les binaires des modules du système d'exécution, répartis entre une bibliothèque `libsx.a` et quelques fichiers "`.o`". Ces derniers sont essentiellement des versions de mise au point de modules existant dans la bibliothèque en version de production. La bibliothèque peut être liée dans un répertoire standard pour permettre l'utilisation de l'option `-lsx` de l'éditeur de liens `ld(1)`.

`$sx/src` contient les sources de tous les modules du système d'exécution. Leur disponibilité permet de les adapter à tous les besoins.

[1]Ada est une marque déposée du Gouvernement des États-Unis d'Amérique (Ada Joint Program Office).

Chapitre 2

BNF : Lecture des Définitions Syntaxiques

2.1 Fonction

BNF lit la grammaire et la met sous forme interne pour utilisation ultérieure par d'autres constructeurs, tels CSYNT et LECL. Il effectue également des tests de cohérence sur la grammaire :

- Chaque règle doit être unique.
- Un non-terminal quelconque doit être productif (c'est-à-dire doit conduire à une chaîne terminale—éventuellement vide).
- Un non-terminal quelconque doit être accessible depuis l'axiome.
- Aucun non-terminal ne doit dériver dans lui-même (c'est-àdire que $N \stackrel{+}{\Rightarrow} N$ est interdit—les *récursions* gauche et droite sont bien entendu autorisées).

BNF construit en outre, pour chaque terminal, la liste des terminaux pouvant le suivre (*contexte*) ; cette information est utilisée par LECL (voir en 4.1.7).

2.2 Métalangage syntaxique

2.2.1 Les éléments syntaxiques

La description de la grammaire utilisée à l'entrée du processeur BNF est très proche de la *forme normale de Backus*—appelée également *forme de Backus-Naur* ("*Backus-Naur Form*", d'où le nom de *BNF*). Toutefois, pour faciliter l'écriture et la lecture de la grammaire, des notations particulières sont employées ; les notations de BNF sont décrites ci-dessous :

- La grammaire est une liste de *règles* appelées aussi *productions*. Le symbole "|" (*ou* de la BNF standard) n'est pas utilisé : il doit y avoir autant de règles que d'alternances.
- Chaque règle se décompose en une *partie gauche* et une *partie droite*, séparées par un caractère "=" (qui remplace le symbole "::=" de la BNF standard), et se termine par un caractère ";".

- Les symboles non-terminaux sont délimités par les caractères “<” et “>” et se composent d'un nombre quelconque de caractères imprimables (autres que “>”) et d'espaces (“␣”).
- Le symbole non-terminal constituant la partie gauche d'une règle doit obligatoirement commencer en colonne 1.
- L'axiome de la grammaire est, par définition, le symbole non-terminal figurant en partie gauche de la première règle.
- On distingue deux types de symboles terminaux :
  - les terminaux *génériques* tels que les identificateurs et les littéraux, pouvant recouvrir un ensemble théoriquement infini de possibilités pour le programme, et qui doivent obligatoirement être définis au niveau lexical (voir le chapitre 4). De tels terminaux sont dénotés par un identificateur (forme C du terme) précédé du caractère “%”.
  - les autres terminaux : mots clés, symboles spéciaux, ...Ceux-ci sont en général écrits exactement tels qu'ils devront apparaître dans les programmes et se terminent au premier blanc ou fin de ligne rencontré. Toutefois, pour résoudre les problèmes d'ambigüité que posent les terminaux commençant par un des caractères

    `; ~ " < % @ & #`

    tout terminal peut être précédé d'un caractère “#”.
    Tout terminal peut également être écrit entre guillemets ; dans ce cas, les conventions d'écriture des chaînes de caractères du langage C s'appliquent— ces conventions sont rappelées dans la table 2.1. La figure 2.1 donne

Tab. 2.1 – Conventions d'écriture des caractères dans les chaînes.

Séquence	Nom du caractère	Code ASCII (en octal)
`\b`	Back-Space	010
`\t`	Horizontal-Tabulation	011
`\n`	New-Line	012
`\f`	Form-Feed	014
`\r`	Carriage-Return	015
`\"`	Double-Quote	042
`\\`	Back-Slash	134
nnn		*nnn*

quelques exemples de possibilités d'écriture de symboles terminaux non-génériques.

Fig. 2.1 – Exemples d'écriture de terminaux non-génériques.

Le terminal	peut s'écrire		
`Begin`	`Begin`	`#Begin`	`"Begin"`
`~`		`#~`	`"~"`
`#`		`##`	`"#"`
`"AND"`		`#"AND"`	`"\"AND\""`
`%AND`		`#%AND`	`"%AND"`
`\`	`\`	`#\`	`"\\"`

– La définition de la grammaire se termine par la fin du fichier d'entrée.

Remarques

– Les règles dont la partie droite est vide sont décrites sous la forme naturelle

```
<A> = ;
```

– On peut inclure des *commentaires* dans la définition de la grammaire : ils débutent en colonne 1 par un caractère "`*`" et se terminent par une fin de ligne.
– Il est possible d'associer de la *sémantique* à chaque règle de grammaire. Sur ce point, BNF est un langage ouvert, au sens où il se contente uniquement d'interpréter les règles de syntaxe d'une description source. BNF considère comme décrivant de la "sémantique" et, à ce titre, ne l'interprète pas, tout le texte se trouvant avant le premier "<" en colonne 1 (marquant le début de la première règle de grammaire), ainsi que les portions de texte situées entre chaque "`;`" marquant la fin d'une règle et le début de la règle suivante (ou la fin du fichier d'entrée). Ce texte peut, bien entendu, être utilisé par un processeur associé à BNF—c'est d'ailleurs ainsi que sont implantés les constructeurs "sémantiques" de SYNTAX : SEMACT, SEMAT, TABC, YAX, ...

2.2.2 Les actions et prédicats

Fondamentalement, la classe de grammaires acceptée par SYNTAX est le LALR(1) (voir en 3.1). Cette classe, bien fondée théoriquement, réalise pour un langage donné un bon compromis entre la rapidité de construction d'un analyseur et la facilité d'écriture de sa grammaire. Cependant, dans certains cas, l'obtention d'une grammaire LALR(1) peut se révéler malcommode—et le résultat obtenu être trop éloigné de la grammaire "naturelle" ; il peut même être impossible d'écrire une telle grammaire (si le langage est non-déterministe). Afin de pallier ces faiblesses, nous avons ajouté à la définition syntaxique deux mécanismes :

– Une spécification de priorité "à la YACC" (voir en 3.1)
– Un mécanisme d'actions et de prédicats décrit ci-dessous. Ces actions et prédicats sont le pendant syntaxique des actions et prédicats de LECL (voir en 4.1.6.3 et 4.1.6.4).

2.2.2.1 Les actions

Il est possible de spécifier des *actions syntaxiques* dans les parties droites des règles de grammaire. Une action syntaxique est un nombre entier n, positif ou nul, précédé du caractère "`@`". Lors de la reconnaissance de l'exécution de l'action "`@`n" pendant l'analyse d'un texte source, un programme écrit par l'utilisateur sera appelé avec le paramètre n. En général ces actions, par utilisation du contexte (gauche ou droit), vont positionner des variables qui peuvent être utilisées par les *prédicats*—voir ci-dessous.

D'un point de vue syntaxique, une action joue le rôle d'un symbole non-terminal dont la définition est fournie par le système—sous la forme d'une partie droite vide. À l'analyse d'un texte l'action n est appelée chaque fois que la règle vide correspondante est reconnue.

L'adjonction d'actions ne change pas le langage décrit par la grammaire ; il est possible en revanche que la grammaire devienne non-LALR(1).

2.2.2.2 Les prédicats

On peut associer un *prédicat* à un symbole terminal quelconque, générique ou non ou à un symbole non-terminal quelconque situé en partie droite d'une règle. Un prédicat est un nombre entier n, positif ou nul, précédé du caractère "&". Un prédicat doit suivre lexicalement dans la grammaire le symbole auquel il est associé. Un tel couple est appelé *terminal étendu* ou *non-terminal étendu*.

À l'analyse, un terminal étendu t&n est reconnu si et seulement si le texte source contient le symbole t et le prédicat n rend *vrai* (le prédicat est implanté par une fonction à résultat booléen, écrite par l'utilisateur, appelée avec le paramètre n). Un non-terminal étendu <A>&n est reconnu si et seulement si une phrase de <A> a été reconnue (on vient d'effectuer une réduction et <A> est en partie gauche de la règle correspondante) et le prédicat utilisateur numéro n a retourné *vrai*. D'un point de vue syntaxique, si s&m et t&n sont des symboles étendus, $\texttt{s\&}m \equiv \texttt{t\&}n \Leftrightarrow \texttt{s} \equiv \texttt{t} \wedge m \equiv n$—ainsi l'on a “"end"&9 ≡ #end␣&009”.

Remarques

- Il est indispensable que le codage des prédicats soit réalisé sans *effet de bord*—les prédicats sont censés être *purs*. En effet, une même occurrence d'un prédicat peut donner lieu à plusieurs exécutions de la fonction qui l'implante (test du "*look-ahead*" puis "*shift*"—pour l'explication de ces termes, voir au chapitre 3—, ou traitement des erreurs…).
- Si, dans un état donné, pour un symbole X, coexistent (*shift* vel *look-ahead*) des occurrences étendues et des occurrences simples, le système considère que les occurrences simples de X dans cet état sont en fait des occurrences étendues X&Else, le prédicat fictif &Else s'évaluant toujours en *vrai*.
- L'ordre d'exécution des prédicats associés à un symbole n'est pas spécifié (sinon que le prédicat fictif &Else est toujours exécuté en dernier). Il faut donc que, dans un état donné, les prédicats associés à un même terminal soient exclusifs les uns des autres.
- Du fait du traitement des erreurs (voir le chapitre 6), lors de l'exécution d'une action, l'unité lexicale suivante est lue et testée (prédicat éventuel compris) et l'unité lexicale d'après est lue et "semi-testée" (il existe dans l'automate une action d'analyse—*Shift* ou *Reduce*—sur ce terminal, mais le prédicat éventuel associé à ce terminal n'est pas encore exécuté). Ceci signifie que la portion de grammaire

  ```
  ... @1 t &1 ...
  ```

  où &1 utilise le résultat de @1 est erronée alors que

  ```
  ... @1 s t &1 ...
  ```

  est valide. Attention : dans cet exemple, l'expression "portion de grammaire" n'implique pas que l'action et le terminal étendu apparaissent dans la même règle.
- Le traitement des erreurs (voir le chapitre 6), et en particulier la phase de correction, est délicat à mettre en œuvre en présence de prédicats. Lorsque le traitement des erreurs a décidé de remplacer la chaîne source $\alpha\beta\gamma$ par

$\alpha\beta'\gamma$ (une erreur a été détectée dans β) il faut obligatoirement que $\alpha\beta'$ soit un préfixe correct d'une sentence du langage. Or β' peut contenir des prédicats qui peuvent s'exécuter différemment dans le contexte de la correction des erreurs (les actions ne sont pas exécutées, l'environnement est différent) et dans le contexte de l'analyse normale et donc produire des résultats différents.

Afin de ne pas construire une chaîne β' qui risquerait d'être invalidée par l'analyse normale—le couple analyseur/correcteur pourrait même boucler indéfiniment—on a choisi de ne pas exécuter les prédicats de l'utilisateur au cours du rattrapage d'erreur, en supposant qu'ils retournent tous *faux*. En procédant de cette façon, il est bien entendu possible d'"oublier" quelques corrections.

Il est donc conseillé d'éviter de détecter des erreurs dans les séquences de prédicats en procédant de la façon suivante. Rappelons que la présence de prédicats dans une grammaire permet d'expliciter d'une manière non syntaxique un choix entre plusieurs possibilités syntaxiques. Considérons, pour simplifier, deux choix C_1 et C_2 reconnus respectivement par les prédicats `&1` et `&2` associés au symbole X. Il y a *a priori* deux façons d'écrire la grammaire correspondante :

- Les symboles étendus X`&1` et X`&2` figurent (en parallèle) dans la grammaire et, en conséquence, l'automate engendré testera successivement `&2`, `&1` (ou inversement) puis `&Else`. Il détectera donc une erreur si ni `&1` ni `&2` n'a retourné *vrai*, erreur qui ne pourra pas être corrigée.
- Un seul choix est explicité (le plus particulier si possible), C_1 par exemple, par le symbole étendu X`&1`, l'autre choix C_2 étant représenté par le symbole simple X qui est donc considéré par le système comme étant associé au prédicat `&Else`. Dans ce cas, l'automate engendré testera successivement `&1` (choix C_1) puis `&Else` (choix C_2). Aucune erreur ne peut donc être détectée à cet endroit. Le choix C_2 a été privilégié. Des erreurs seront détectées ultérieurement si la chaîne ne correspond ni au choix C_1 ni au choix C_2 et les corrections correspondantes tenteront de la rapprocher du choix C_2. Le lecteur pourra regarder à ce propos les grammaires des exemples suivants.

On trouvera en annexe B.1 la grammaire syntaxique décrivant le langage LECL. Cette grammaire utilise un prédicat syntaxique (`&1`) associé à un terminal générique pour traiter la suite de mots clés non réservés NOT" KEYWORD" (sinon, la grammaire proposée n'est pas LR(1)). On trouvera en annexe B.4 le programme C codant le prédicat `&1`. On trouvera en annexe E.3 une grammaire LALR(2) où les conflits LALR(1) sont résolus par l'utilisation d'un non-terminal étendu et le programme codant le prédicat correspondant. On trouvera de même en annexe E.4 la grammaire d'un langage non-déterministe et le programme associé réalisant les actions et prédicats syntaxiques nécessaires.

2.3 Mise en œuvre

Se reporter à la documentation en ligne—`man bnf`.

2.4 Exemples de définitions syntaxiques

On trouvera en annexe A.1 la grammaire du processeur BNF lui-même et en annexe B.1 la grammaire de LECL (voir le chapitre 4).

Chapitre 3

CSYNT : Construction de l'Analyseur Syntaxique

3.1 Classe de grammaires acceptée

CSYNT accepte en entrée des grammaires de la classe LALR(1) ; ceci permet à notre avis de réaliser le meilleur compromis possible à l'heure actuelle entre le temps de construction d'une part et la puissance d'expression permise par la classe de grammaires acceptée d'autre part.

CSYNT reçoit en entrée la forme interne de la grammaire produite par BNF (voir le chapitre 2) et construit un analyseur syntaxique ascendant sous forme d'un automate à pile.

Si CSYNT, au cours de cette construction, détecte une non-conformité dans la définition de la grammaire, il le signale de deux façons :

- Le message `NON LALR(1)` est écrit sur `stderr` (en général le terminal de l'utilisateur).
- Un diagnostic circonstancié des causes de cette non-conformité est inscrit dans le fichier *nom-du-langage*`.la.l`, donnant notamment les règles de la grammaire qui la produisent et une partie du contexte.

3.2 Les conflits

On peut avoir deux types de non-conformité :

- des conflits *Reduce/Reduce* : il existe plusieurs possibilités de reconnaissance simultanée de parties droites de règles (*Reduce*) et les symboles du contexte droit de ces règles (*look-ahead*) ne permettent pas de faire un choix.
- des conflits *Shift/Reduce* : il y a conflit entre l'absorption (*shift*) du symbole suivant, et une réduction—ou plusieurs.

Dans les messages d'erreur émis par CSYNT, chacune des règles de grammaire mentionnées comporte un *marqueur LR*, symbolisé par un "`.`", qui indique où a été détectée la non-conformité.

Exemple : La figure 3.1 donne un exemple de diagnostic émis par CSYNT lors de la construction d'un analyseur.

FIG. 3.1 – Exemple de diagnostic de non-conformité *Shift/Reduce*.

```
NON LALR(1)
Le terminal t est en conflit dans l'etat s

    - en position reduce dans :

    p1: <A> = alpha .
            derivant de :
            p2: <B> = beta . <D> gamma

    - en position shift dans :

    p3: <C> = delta . t lambda
```

La signification de ce diagnostic est la suivante : on vient de reconnaître, dans l'état `s`, simultanément `alpha` (partie droite de `p1`) et `delta` (début de `p3`) et l'on doit décider s'il faut effectuer la réduction `p1` (*Reduce*) ou lire le symbole `t` (*Shift*) de `p3` ; or l'on a `<D>` $\stackrel{*}{\Rightarrow}$ `omega <A>` et `t` $\in$ `FIRST1(gamma)`.

La résolution du conflit devant se faire avec (au plus) un symbole en avance, la vue de `t` ne permet donc pas de prendre de décision, d'où le message de non-conformité.

On trouvera en annexe E.1 une grammaire ambigüe illustrant le problème du "dangling else" et en annexe E.1.2.1 les messages de non-conformité produits par CSYNT.

Si les renseignements précédents ne suffisent pas à comprendre les raisons d'un conflit, L'option `-path` du processeur CSYNT (voir la documentation en ligne—`man csynt`) permet d'obtenir des renseignements supplémentaires.

Pour chaque état `s` dans lequel un conflit est détecté, CSYNT sort une chaîne $X_1 \ldots X_i \ldots X_n$ où chaque X_i est un élément du vocabulaire (terminal ou non terminal) telle que l'état `s` est atteint depuis l'état initial par transition sur cette chaîne. Cette chaîne permet de comprendre pourquoi les productions mises en jeu dans un conflit sont atteintes simultanément.

Pour chaque terminal `t` en conflit dans un état `s`, outre les renseignements standards décrits précédemment, CSYNT sort pour chaque production

```
p1: <A> = alpha .
```

impliquée dans le conflit et pour chaque item

```
p2: <B> = beta . <D> gamma
```

"responsable" du contexte droit `t` (`t` $\in$ `FIRST1(gamma)`), une dérivation droite, issue de l'axiome `<S>` de la grammaire de la forme :

```
<S>
 =*=> delta <B> ...
 =*=> delta beta <D> gamma ...
 =*=> delta beta <D> t ...
```

$\vdots$
$\stackrel{*}{\Rightarrow}$ `delta beta omega <A> t ...`
$\stackrel{*}{\Rightarrow}$ `delta beta omega alpha t ...`

qui montre le "cheminement" complet du terminal `t` depuis son apparition comme débutant de la chaîne `gamma` jusqu'à son utilisation comme contexte droit de la production `p1`.

L'option `-path` demande également au constructeur CSYNT de rechercher certain cas d'ambigüité (le cas général est indécidable). Si CSYNT détecte une ambigüité, deux dérivations droites différentes menant à la même forme sentencielle sont également produites.

On trouvera en annexe E.1.2.2 les messages de non conformité produits avec l'option `-path` sur l'exemple de l'annexe E.1.

Pour un conflit donné, si aucune ambigüité a été détectée, l'option `-path` demande également à CSYNT de vérifier si la grammaire est LR(1). Si la grammaire n'est pas LR(1), CSYNT produit deux dérivations droites différentes qui violent la définition du LR(1).

Première dérivation :

`<S>`
$\stackrel{*}{\Rightarrow}$ `beta <A> t z`
$\Rightarrow$ `beta alpha t z`

Deuxième dérivation :

`<S>`
$\stackrel{*}{\Rightarrow}$ `gamma`
$\Rightarrow$ `beta alpha t z'`

avec

`gamma` $\neq$ `beta <A> t z'`

Outre un intérêt pédagogique évident, ces deux dérivations peuvent aider à la compréhension du conflit.

On trouvera en annexe E.2 une grammaire non LR(1) et les messages de non-conformité produits par CSYNT lorsque l'option `-path` est positionnée.

À des fins pédagogiques ou pour comprendre les raisons de conflits complexes, l'option `-lalr1` de CSYNT permet d'obtenir l'automate LALR(1) sous une forme lisible : les items constituant chaque état sont explicités, les transitions (avant et arrière) reliant entre eux les états sont indiquées ainsi que les ensembles de symboles terminaux formant le contexte droit (*look-ahead*) de chaque réduction impliquée dans un conflit LR(0).

3.3 Leur résolution

Face à une grammaire présentant des conflits, on peut réagir de quatre façons (non exclusives) :

1. Supprimer les conflits en transformant la grammaire afin de la rendre LALR(1).
2. Ne rien faire car les règles par défaut choisies par le constructeur conviennent.
3. Écrire une spécification "à la YACC" qui indique au constructeur les choix de l'utilisateur face à un conflit.

4. Insérer dans la grammaire des prédicats *vel* des actions (voir en 2.2.2) afin d'aider à la résolution des conflits.

Dans tous les cas, sur un conflit, avant de choisir un type de résolution, il faut comprendre la raison profonde de ce conflit. Cette tâche peut ne pas être facile (voir le chapitre précédent et la documentation en ligne—`man csynt`).

Chacune de ces méthodes a ses avantages et ses inconvénients :

1. C'est la méthode des "purs et durs". Outre le fait qu'elle puisse être délicate à mettre en œuvre, elle a tendance à éloigner la grammaire de sa forme "naturelle". Il peut être parfois préférable de décrire un sur-langage, le complémentaire étant filtré par la "sémantique".
2. Par défaut, le système donne priorité :
   – au *Shift* lors d'un conflit *Shift/Reduce*,
   – à la règle qui apparaît la première dans la grammaire, lors d'un conflit *Reduce/Reduce*.
3. Lors de la détection du premier conflit, CSYNT cherche un fichier de nom *nom-du-langage*`.dt`. Si ce fichier existe, il est censé contenir des tables, produites par le module PRIO, qui vont aider le constructeur CSYNT à résoudre les conflits. Voir ci-dessous le format et la signification des spécifications destinées à PRIO.

   L'avantage de cette façon de procéder est la grande finesse possible pour la résolution des conflits, l'inconvénient en est que l'on peut ne plus savoir quel est le langage reconnu...
4. L'utilisation de prédicats permet d'influer sur l'analyse et en particulier de faire intervenir des notions qui n'appartiennent pas à la classe LALR(1). Ces notions peuvent être purement syntaxiques—utilisation du contexte (gauche *vel* droit) sur une longueur éventuellement non bornée—ou même "sémantiques" (utilisation d'informations calculées par les actions syntaxiques).

   Les prédicats peuvent représenter le seul moyen utilisable pour résoudre des conflits—voir l'exemple du langage non-déterministe en pages 87 et suivantes. Ils ont cependant pour inconvénient de ne résoudre les conflits que de manière *dynamique*. Ils ne se substituent donc pas aux autres possibilités énumérées ci-dessus.

3.3.1 Le langage PRIO

Les tables mentionnées ci-dessus sont construites par l'exécution préalable du module PRIO (voir la documentation en ligne—`man prio`) sur des spécifications de priorités, écrites par l'utilisateur et rassemblées dans un fichier de nom *nom-du-langage*`.prio`.

Une spécification écrite dans le langage PRIO contient deux parties, chacune étant optionnelle :

– une partie (a) où l'on donne les priorités des opérateurs ;
– une partie (b) où l'on donne les priorités des règles de la grammaire.

(a) Cette partie définit, pour chaque symbole cité (en général un terminal de la grammaire), son type d'associativité (*gauche*, *droit*, *non associatif*) et sa priorité relative.

Cette liste est ordonnée du moins prioritaire vers le plus prioritaire. Ainsi

```
%left + -
%left * /
```

décrit la priorité et l'associativité des quatre opérations arithmétiques usuelles. L'addition et la soustraction sont associatives à gauche et sont moins prioritaires que la multiplication et la division qui sont elles aussi associatives à gauche.

Le mot clé `%right` est utilisé pour définir l'associativité droite et le mot clé `%nonassoc` est utilisé pour définir des opérations qui, comme le `.LT.` de FORTRAN, ne peuvent s'associer avec elles-mêmes. Ainsi

```
A .LT. B .LT. C
```

est illégal en FORTRAN.

(b) En règle générale, une production prend la priorité et l'associativité de son terminal le plus à droite (s'il existe). Il est possible de modifier cette règle générale pour une production, en écrivant cette production et en la faisant suivre du mot clé `%prec` suivi d'un symbole. La production prend alors la priorité et l'associativité de ce symbole. Ce symbole (et donc sa priorité) doit avoir été défini dans la partie (a).

Exemple : Considérons la grammaire de la figure 3.2. Cette grammaire est

FIG. 3.2 – Grammaire d'expressions.

```
<EXPR>  = <EXPR> + <EXPR>   ;
<EXPR>  = <EXPR> - <EXPR>   ;
<EXPR>  = <EXPR> * <EXPR>   ;
<EXPR>  = <EXPR> / <EXPR>   ;
<EXPR>  = - <EXPR>          ;
<EXPR>  = %IDENT            ;
```

de toute évidence ambigüe ; il est cependant possible de résoudre aisément les conflits, comme le montre la figure 3.3. La dernière règle de cette spécification

FIG. 3.3 – Désambigüation de la grammaire d'expressions.

```
%left + -
%left * /

<EXPR>  = - <EXPR>          ; %prec *
```

indique que la priorité et l'associativité de la production citée (donc du "`-`" unaire) sont celles de "`*`".

Les règles de *désambigüation* sont utilisées par CSYNT de la façon suivante pour résoudre des conflits :

1. On enregistre les précédences et associativités des terminaux (s'il y en a).
2. Une précédence et une associativité sont associées à chaque règle de la grammaire : ce sont la précédence et l'associativité du symbole terminal le plus à droite dans cette règle (s'il existe). Si la construction `%prec` est

utilisée, la valeur spécifiée vient remplacer la valeur par défaut. Certaines règles de grammaire et certains terminaux peuvent donc ne pas avoir de priorité *vel* d'associativité.

3. Quand un conflit *Shift/Reduce* ou *Reduce/Reduce* se produit, si le symbole en conflit ou une règle impliquée dans un *Reduce* n'a ni priorité ni associativité, alors les règles par défaut (voir page 18) s'appliquent et le conflit est signalé.
4. Sinon le conflit est résolu en faveur de l'action (*Shift* ou *Reduce*) associée à la précédence la plus forte. Si les précédences sont identiques, alors l'associativité est utilisée : associativité *gauche* implique *Reduce*, associativité *droite* implique *Shift* et non-associativité implique *erreur*. Si le conflit ne peut être résolu malgré tout, on applique les règles par défaut.

La plus grande prudence est vivement conseillée dans l'emploi des règles de désambigüation.

3.4 Optimisation de l'automate

Les automates produits par le constructeur LALR(1) sont optimisés lors d'une deuxième phase nommée OPTIM. Les optimisations effectuées sont de deux ordres :

optimisations propres à l'analyse LR : L'automate est tout d'abord partitionné en trois : une partie qui traite les transitions terminales (les "T-tables"), une autre s'occupant des transitions non-terminales (les "NT-tables") et une troisième partie (les "Prdct-tables") qui gère le traitement des prédicats.

Les automates correspondants sont réduits. Les productions simples *sans sémantique* sont partiellement éliminées (cette élimination peut être totale, sur option : voir la documentation en ligne—`man csynt`), les transitions non-terminales (après réduction) ne nécessitant pas la pile d'analyse sont détectées et les NT-tables sont compactées.

techniques de représentation de matrices creuses : Les T-tables et les NT-tables sont linéarisées de façon à conserver un temps constant pour l'accès à l'information.

Cette forme des tables est bien évidemment inexploitable manuellement. L'utilisateur peut cependant avoir une idée de l'automate à pile engendré, en utilisant l'option `-floyd_evans` (voir la documentation en ligne—`man csynt`) qui permet d'obtenir le listage de l'automate sous une forme compréhensible.

3.5 Listage de l'automate engendré

L'automate à pile engendré est partitionné en trois tables :
- les "T-tables" qui s'occupent des transitions terminales,
- les "NT-tables" qui s'occupent des transitions non-terminales,
- les "Prdct-tables" qui s'occupent du traitement des prédicats.

Chacune de ces tables est partitionnée en *blocs*, comprenant une séquence d'*instructions*. Chaque bloc est étiqueté pour référence, de la façon suivante :

T-bloc `Etq` l:

NT-bloc `State` m:

Prdct-bloc `Prdct` n:

À chaque instruction est associée une adresse (nombre entier positif). Une instruction est formée d'un doublet (test, action). Le premier champ sert à la sélection, le second champ indique l'action à effectuer lorsque l'instruction a été sélectionnée.

L'analyse commence dans les T-tables, par *activation* d'un T-bloc dont l'étiquette est précisée dans le listage. Le symbole terminal sous la tête de lecture est le symbole (fictif) `End Of File`.

Quand un bloc est activé, une instruction et une seule de sa séquence est sélectionnée et exécutée : les instructions sont examinées, l'une après l'autre, jusqu'à en rencontrer une dont le *test* soit satisfait. La dernière instruction d'une séquence contient toujours un champ *test* valant `Any`, ce qui assure une sélection inconditionnelle de cette instruction.

Le mode de sélection dépend de la table considérée :

- Une instruction d'un T-bloc est sélectionnée si et seulement si le symbole terminal sous la tête de lecture est celui du champ *test*.
- Une instruction d'un NT-bloc est sélectionnée si et seulement si le symbole non-terminal courant, c'est-à-dire le non-terminal en partie gauche de la production qui vient d'être reconnue et choisie, est celui du champ *test*.
- Une instruction d'un Prdct-bloc est sélectionnée si et seulement si la fonction utilisateur appelée avec le paramètre *test* retourne *vrai*.

Lorsqu'une instruction est sélectionnée, l'action spécifiée par le deuxième champ est exécutée. Il existe quatre types d'actions : *Shift*, *Reduce*, *Halt* et *Error*. Chacun de ces types est décrit ci-dessous :

L'action Shift correspond au traitement d'une transition (terminale ou non-terminale) ; elle est codée sur trois champs :

Scan	Stack	Goto

Si le champ *Scan* contient "`*`", le symbole terminal suivant du texte source est positionné sous la tête de lecture ; sinon, ce champ est vide et aucune lecture n'est effectuée.

Le champ *Stack*, obligatoire, contient une information qui est empilée sur la pile d'analyse. Ce peut être soit un nombre entier n positif entre parenthèses, soit un nombre entier n, positif ou nul.

- (n) signifie que n est le numéro d'un NT-bloc ;
- n signifie que n est l'adresse d'une instruction du NT-bloc. Cette instruction, apres une action *Reduce*, pourra être exécutée inconditionnellement. Si n est nul, cette adresse n'est pas significative.

Le champ *Goto*, obligatoire, indique le prochain T-bloc qui va être activé (`Goto Etq` l).

L'action Reduce correspond au traitement d'une réduction ; elle est codée sur quatre champs :

Scan	Reduce	Stack	Goto

Si le champ *Scan* contient "`*`", le symbole terminal suivant du texte source est positionné sous la tête de lecture ; sinon, ce champ est vide et aucune lecture n'est effectuée.

Le champ *Reduce* contient un nombre entier positif p qui est le numéro de la règle de grammaire qui vient d'être reconnue et choisie.

Le champ *Stack* contient un certain nombre (éventuellement nul) de barres verticales “|”, qui indiquent le nombre dont la pile d'analyse est écrêtée. Ce nombre est, pour une instruction d'un T-bloc, la longueur de la partie droite de la production p, cette longueur diminuée de un pour un NT-bloc. Le champ *Goto* contient soit un non-terminal `<nt>` spécifié par

```
Goto Branch_Table (<nt>, Top_Stack_State)
```

soit une adresse `n` spécifiée par

```
Goto n
```

Ce champ est interprété de la façon suivante : après avoir écrêté la pile d'analyse de la longueur spécifiée par le champ *Stack*, on examine le contenu du champ *Goto*. Si c'est `Goto` n, on va exécuter inconditionnellement l'instruction numéro n. Sinon, on examine le sommet de la pile d'analyse. Si c'est l'adresse d'une instruction (voir action *Shift*), on va exécuter inconditionnellement cette instruction (on ne peut jamais trouver l'adresse zéro). Sinon, le sommet de pile désigne un NT-bloc qui est activé et l'on recherche dans sa liste d'instructions le non-terminal spécifié dans le champ *Goto*. Ce non-terminal `<nt>` est en général le non-terminal se trouvant en partie gauche de la production numéro p (du champ *Reduce*) mais, du fait des optimisations, ce peut être un autre non-terminal ou même un non-terminal “inventé” par l'optimiseur (dénoté par un nombre entier n compris entre un “>” et un “<”—donc “>n<”).

L'action Halt marque la fin de la reconnaissance du texte.

L'action Error indique la détection d'une erreur de syntaxe. Elle provoque l'exécution d'un programme de rattrapage d'erreur (voir le chapitre 6).

En fait, les instructions qui viennent d'être décrites sont une forme simplifiée d'un langage permettant de coder des automates à pile déterministes et dont les instructions sont appelées *productions de Floyd-Evans*.

Une production de Floyd-Evans comporte cinq champs :

Test	Scan	Reduce	Stack	Goto

Il y a dix types différents de productions de Floyd-Evans :

Test	Scan	Reduce	Stack	Goto
test	*		(n° de NT-bloc)	n° de T-bloc
test	*		adresse	n° de T-bloc
test			(n° de NT-bloc)	n° de T-bloc
test			adresse	n° de T-bloc
test	*	n° de réduction	|···	adresse
test	*	n° de réduction	|···	non-terminal
test		n° de réduction	|···	adresse
test		n° de réduction	|···	non-terminal
test				`Halt`
test				`Error`

Remarque : Afin de faciliter l'exploitation manuelle de l'automate, le code des NT-tables est présenté dans le listage successivement sous deux formes :

1. La première partie, organisée comme on l'a présenté ci-dessus, regroupe séquentiellement les instructions de chaque NT-bloc. Cette présentation

n'est exploitée que lorsque le sommet de pile, apres écrêtage, représente un numéro de NT-bloc et le champ *Goto* de l'instruction spécifie un non-terminal.

2. Dans la seconde présentation, qui, elle, est exhaustive, les instructions sont repérées par leur numéro et sont exécutées inconditionnellement ; c'est la partie qui doit être utilisée lorsque le champ *Goto* d'une instruction *Reduce* est `Goto`n ou lorsque le sommet de pile après écrêtage est une adresse.

3.6 Mise en œuvre

Se reporter à la documentation en ligne—`man bnf`.

Chapitre 4

LECL : Construction de l'Analyseur Lexical

LECL est le constructeur lexical de SYNTAX. Nombre de ses caractéristiques ont été empruntées à son prédécesseur *new_10cl*, mais il s'en distingue notamment par une plus grande facilité de description et une plus grande puissance de spécification.

LECL a pour données, d'une part la description des unités lexicales du langage sous forme d'expressions régulières, fournie par l'utilisateur, et d'autre part des informations extraites des tables produites par BNF (*nom-du-langage*.bt) ; son rôle est de produire un automate d'états finis capable d'effectuer la reconnaissance des terminaux du niveau syntaxique.

4.1 Métalangage lexical

La spécification d'un analyseur lexical en LECL est un texte qui doit vérifier une certaine syntaxe. Le langage LECL peut donc se décrire à l'aide du système SYNTAX—voir l'annexe B.

Informellement, le niveau lexical de LECL se définit comme suit :

- C'est un langage non positionnel (le placement d'une unité lexicale dans une ligne est sans importance) qui comporte des mots clés réservés.
- Les mot clés de LECL, au nombre de 43, sont répertoriés dans la table 4.1.

  Les mots clés commençant par "&" représentent les *prédicats système* (voir en 4.1.6.4), ceux commençant par "@" représentent les *actions système* (voir en 4.1.6.3).
- L'ensemble des caractères ASCII est licite.
- Les identificateurs sont formés d'une lettre suivie d'un nombre quelconque (éventuellement nul) de lettres ou de chiffres pouvant être précédés d'un blanc souligné ("_").
- La capitalisation des lettres n'a aucune influence sur la signification des identificateurs ou des mots clés.

TAB. 4.1 – Les mots clés de LECL.

&FALSE	&IS_FIRST_COL	&IS_LAST_COL
&IS_RESET	&IS_SET	&TRUE
@DECR	@ERASE	@INCR
@LOWER_CASE	@LOWER_TO_UPPER	@MARK
@PUT	@RELEASE	@RESET
@SET	@UPPER_CASE	@UPPER_TO_LOWER
ABBREVIATIONS	ALL	BITS
BUT	BYTE	CLASSES
CODE	COMMENTS	CONTEXT
END	END_OF_FILE	EOF
FOR	INCLUDE	INTERNAL
PRIORITY	REDUCE	SHIFT
SYNONYMS	TOKENS	UNBOUNDED
UNION	UNUSED	USE
WORD		

– Certains identificateurs peuvent représenter des terminaux du niveau syntaxique où la capitalisation est significative. Si ces terminaux comportent des lettres minuscules, il faut les représenter comme des chaînes de caractères (entre guillemets) en conservant la capitalisation utilisée dans BNF.
– Un commentaire est introduit par deux signes moins ("`--`") et se termine à la première fin de ligne rencontrée.
– Les chaînes de caractères sont représentées entre guillemets ("`"`"). Ces guillemets délimitent un nombre strictement positif de caractères quelconques dont la convention de représentation est celle du langage C—voir en page 10.

La définition syntaxique du langage est divisée en cinq parties. Chacune de ces parties est optionnelle.

4.1.1 Les classes

Le vocabulaire terminal du niveau lexical est généralement un sous-ensemble des caractères valides pour une machine donnée : ASCII, EBCDIC...

Cependant, l'utilisation directe et systématique des caractères comme vocabulaire terminal des expressions régulières peut
– être malcommode pour l'utilisateur
– engendrer un grand nombre d'états pour l'automate
d'où l'introduction de la notion de *classe*.

Une classe est un ensemble de caractères qui jouent un rôle identique à un moment donné.

Exemple : Dans la définition des identificateurs, les lettres jouent un rôle analogue ; d'un point de vue lexical le fait important est d'avoir une lettre et non de connaître son nom. Dans un identificateur l'ensemble des lettres majuscules ou minuscules peut donc constituer une *classe*.

Il existe un certain nombre de classes prédéfinies en LECL. Le nom de ces

classes n'est pas réservé, l'utilisateur peut donc les redéfinir s'il le désire. La table 4.2 liste l'ensemble des classes qui ont été prédéfinies pour simplifier l'écriture des spécifications LECL, et la table 4.3 donne les noms qui sont associés aux caractères non imprimables du code ASCII.

TAB. 4.2 – Classes prédéfinies pour le confort de l'utilisateur.

Nom	Signification
`ANY`	Ensemble des caractères ASCII
`EOL` ou `NL`	Fin de ligne (code octal 012)
`LETTER`	Les lettres, majuscules et minuscules
`LOWER`	Les lettres minuscules
`UPPER`	Les lettres majuscules
`DIGIT`	Les chiffres décimaux
`QUOTE`	Caractère "`"`" (Code octal 042)
`A`	"`Aa`"
`B`	"`Bb`"
⋮	⋮
`Z`	"`Zz`"

Une classe est *nommée* si l'ensemble des caractères qui la composent a reçu un nom. Cette définition est l'objet de la première partie d'un source LECL, dans laquelle on définit les classes nommées en regroupant tous les caractères les composant.

Cette définition débute par le mot clé `CLASSES`.

Une classe se définit par un nom (*identificateur*), suivi du symbole "`=`", suivi d'une expression ; la définition se termine par un "`;`".

L'expression utilisée admet

comme opérandes :
- des classes prédéfinies ;
- des chaînes de caractères (l'ensemble des caractères de la chaîne appartient à la classe) ;
- des codes octaux (nombre octal de trois chiffres précédé du caractère "`#`"), dont la valeur est la représentation interne en ASCII du caractère désigné ;
- des intervalles décrivant une suite de caractères dont les codes internes sont contigus ; un intervalle est représenté par le premier et le dernier caractère de la séquence, séparés par "`..`". Ces caractères marquant le début et la fin de la séquence sont représentés, soit par le caractère entre guillemets, soit par son code interne exprimé en octal et précédé du caractère "`#`", soit par le nom (s'il existe) de ce caractère. L'ordre sous-jacent est celui de l'ASCII et le code interne du premier caractère d'un intervalle doit bien entendu être inférieur ou égal à celui du second.
- des classes nommées définies préalablement ;

comme opérateurs (binaires infixes) :
- l'union ensembliste : "`+`" ;
- la différence ensembliste : "`-`".

TAB. 4.3 – Classes prédéfinies pour les caractères non-imprimables.

Nom	Code ASCII (octal)	
NUL	000	
SOH	001	
STX	002	
ETX	003	
EOT	004	
ENQ	005	
ACK	006	
BEL	007	
BS	010	Back-Space
HT	011	Horizontal-Tabulation
LF	012	Line-Feed
VT	013	Vertical-Tabulation
FF	014	Form-Feed
CR	015	Carriage-Return
SO	016	
SI	017	
DLE	020	
DC1	021	
DC2	022	
DC3	023	
DC4	024	
NAK	025	
SYN	026	
ETB	027	
CAN	030	
EM	031	
SUB	032	
ESC	033	
FS	034	
GS	035	
RS	036	
US	037	
SP	040	Space
DEL	177	

Une classe est *anonyme* si sa dénotation apparaît directement à l'endroit de son utilisation (par exemple dans une expression régulière).

Tous les caractères licites doivent être définis. Cette définition peut être explicite ou implicite. Un caractère est défini explicitement s'il apparaît dans une définition de classe (nommée ou anonyme) ; il est défini implicitement s'il intervient dans le nom d'un terminal non-générique du niveau syntaxique alors qu'il n'est pas explicitement défini.

Tout caractère non défini (implicitement ou explicitement) est un caractère *interdit* dans le langage.

FIG. 4.1 – Exemple de définition de classes.

```
Classes
   tout_sauf_EOL   = ANY - EOL ;
   tout_sauf_quote = ANY - quote ;
   printable       = ANY - (#000..SP + #177) ;
```

4.1.2 Les abréviations

Afin d'éviter l'écriture fastidieuse de sous-expressions régulières identiques, il est possible de nommer de telles expressions et d'utiliser ce nom dans une expression régulière ultérieure. Toute occurrence du nom d'une expression régulière est sémantiquement équivalente à une occurrence parenthèsée de l'expression qu'elle dénote.

Cette rubrique débute par le mot clé `ABBREVIATIONS`.

Une abréviation se définit par un nom (*identificateur*), suivi du symbole "`=`", suivi d'une expression régulière (voir en 4.1.6) ; la définition est terminée par un "`;`".

On peut également définir dans cette partie des expressions de prédicats (voir en 4.1.6.4).

4.1.3 Les unités lexicales

La définition lexicale se poursuit par la définition des expressions régulières qui permettent de reconnaître les différentes unités lexicales du langage.

Cette définition débute par le mot clé `TOKENS`.

Une définition d'unité lexicale s'écrit dans le cas le plus simple de la manière suivante : un nom d'unité lexicale, suivi du symbole "`=`", suivi d'une expression régulière terminée par un "`;`" ; ce "`;`" peut être éventuellement suivi d'une spécification de priorité, de contexte ou de post-action (voir en 4.1.7 et 4.1.6.3).

Un nom d'unité lexicale peut être :

- le mot clé `COMMENTS` ;
- le mot clé `INCLUDE` ;
- un terminal générique (voir en 2.2.1) ;
- une chaîne de caractères (terminal du langage entre guillemets) ;
- le mot clé `EOF` (pour "*E*nd *O*f *F*ile", voir plus loin) ;
- un identificateur (dont le nom n'est pas significatif).

L'unité lexicale de nom "COMMENTS" définit les *commentaires*. L'expression régulière de sa partie droite doit décrire non seulement les commentaires du langage mais également les *espacements* entre les symboles terminaux (blanc, fin de ligne, *et cetera*).

L'unité lexicale de nom "INCLUDE" permet de définir des *includes* (voir en 4.1.6.5).

Les terminaux génériques sont les seuls qui doivent obligatoirement être définis au niveau lexical. Les autres terminaux sont traités par LECL de la façon suivante :

> Pour un terminal donné non défini (c'est-à-dire dont le nom n'apparaît pas en partie gauche d'unité lexicale), LECL regarde s'il est reconnu par l'expression régulière d'une unité lexicale :
> – si non : LECL engendre automatiquement l'expression régulière le reconnaissant, caractère par caractère.
> – si oui : ce terminal est rangé dans une table qui sera utilisée par l'*analyseur lexical*.

Lors de l'analyse lexicale, si une chaîne de caractères est reconnue par une expression régulière, plusieurs cas peuvent se produire :

- L'unité lexicale est "EOF" (*End Of File*) : l'analyseur lexical renvoie le *code* de "EOF" à l'analyseur syntaxique.
- L'unité lexicale est un terminal : l'analyseur lexical renvoie le code du terminal et mémorise la chaîne de caractères dans une table. Cette table est manipulable par l'intermédiaire des procédures du "string_manager" (voir la documentation en ligne—man sxstr_mngr).
- L'unité lexicale est "COMMENTS" : les caractères composant cette unité sont accessibles (s'ils ont été conservés) par l'intermédiaire de la structure "sxterminal_token" (voir la documentation en ligne—man sxunix) associée à l'unité lexicale qui suit ces commentaires.

Exemple : Dans la définition syntaxique on trouve la règle :

```
<expression>   =   <expression>   PLUS   <terme>   ;
```

et au niveau lexical, on trouve :

```
PLUS    =  "+" ;
```

"PLUS" est un terminal du langage. Lorsque l'analyseur lexical reconnaît la chaîne "+", il renvoie le code du terminal "PLUS".

Remarque : L'unité lexicale "EOF" permet à l'utilisateur de préciser que ses fichiers se terminent par une certaine chaîne de caractères. Si l'on ne mentionne pas EOF dans les unités lexicales, le système génèrera un EOF lors de la reconnaissance de la fin physique du fichier.

4.1.3.1 Les mots clés

Par définition un *mot clé* est un terminal non générique qui est reconnu par une unité lexicale dont le nom est différent de celui du mot clé.

Exemple :

```
%ID = LETTER {LETTER | DIGIT} ;
```

Supposons que le terminal "`begin`" apparaisse dans la grammaire syntaxique, mais ne soit pas décrit au niveau lexical. La chaîne de caractères "`begin`" étant reconnue par `%ID`, le terminal "`begin`" est par définition un mot clé.

Lecl construit le texte en langage C d'une fonction de nom `check_keyword` qui permet de vérifier si une chaîne est ou non un mot clé. Le texte de cette fonction fait partie intégrante des tables d'analyse produites par le module Tables_C—voir le chapitre 7. Chaque fois qu'un terminal est reconnu par une unité lexicale pouvant reconnaître des mots clés, l'analyseur lexical appelle `check_keyword` qui retourne :

- 0 : le terminal n'est pas un mot clé ; le code retourné par l'analyseur lexical est celui de l'unité lexicale (`%ID` dans l'exemple) ;
- $n > 0$: le terminal est un mot clé de code interne n.

Si l'unité lexicale est un identificateur qui n'est le nom d'aucun terminal du langage, l'analyseur lexical effectue la même recherche en table que dans le cas précédent. Si cette recherche réussit, l'analyseur lexical retourne à l'analyseur syntaxique le code du terminal (mot clé) ainsi reconnu. Dans le cas contraire, il élimine purement et simplement la chaîne reconnue (en signalant une erreur) et ne renvoie pas de code. Ce genre d'unité lexicale permet donc de décrire un ensemble de terminaux du langage par une expression régulière unique, dans le cas où ils ne peuvent pas être reconnus par un terminal générique.

Exemple : Dans le langage Lecl, les séquences de caractères `@DECR`, `@ERASE`, ... sont des mots clés réservés qui désignent des actions prédéfinies. Dans la version actuelle du langage ces mots clés sont au nombre de 12. De plus, toute autre séquence de caractères vérifiant l'expression régulière

```
"@" LETTER {["_"] LETTER}
```

est erronée. La description des ces mots clés est réalisée simplement par l'ajout de la définition

```
KEYWORD = "@" LETTER {["_"] LETTER};
```

où `KEYWORD` est un identificateur qui n'est pas un symbol terminal de la grammaire du niveau syntaxique du langage Lecl.

De plus, le source d'une spécification Lecl a pu être créé par un programme (paragrapheur ou autre) qui a produit les mots clés en gras. Chaque caractère `"c"` d'un mot clé a pu être superposé un certain nombre de fois sur lui-même (par la séquence `"c" {BS "c"}+` pour créer une impression de gras. Chaque mot clé peut donc maintenant avoir un nombre infini de représentations dont la reconnaissance, si on adopte la philosophie pronée par Lecl (dégrossissage de la reconnaissance par une expression régulière puis utilisation de la fonction `check_keyword` générée automatiquement) nécessite l'obtention d'une forme normale. Or Lecl a la possibilité de modifier la chaîne source (voir en 4.1.6.1 et 4.1.6.3) pour obtenir une forme interne qui sera réellement utilisée. Ainsi les expressions régulières

```
DARK_A   = "A" -{BS "A"}+ | "a" -{BS "a"}+ ;
DARK_B   = "B" -{BS "B"}+ | "b" -{BS "b"}+ ;
           .
           .
DARK_LET = DARK_A | DARK_B | ... ;
```

permettent de définir des lettres grasses dont la forme normale contient une lettre unique. L'expression régulière

```
DARK_WORD = DARK_LET {[DARK_US] DARK_LET} ;
```

où

```
DARK_US = "_" -{BS "_"}+ ;
```

décrit un identificateur dont la forme externe est "grasse" et dont la forme normale est épurée.

Il serait alors agréable que la définition

```
KEYWORD = "@" DARK_WORD ;
```

permette automatiquement de reconnaître les mots clés désignant les actions système (et de générer la fonction `check_keyword`). Or du fait de la présence de l'opérateur "+" dans la définition d'une lettre grasse, chaque lettre est répétée au moins deux fois et ne correspond donc pas à la forme normale bien que la forme interne obtenue après application de l'opérateur "-" soit la forme normale du mot clé. Pour que l'utilisation précédente soit possible, la définition des mots clés a été (légèrement) modifiée :

Un mot clé est un terminal non générique qui :

- soit est reconnu par une unité lexicale dont le nom est celui d'un terminal (générique ou non) de la grammaire du niveau syntaxique, différent de celui du mot clé ;
- soit est la forme interne obtenue après reconnaissance et application (éventuelle) des opérateurs "-" par une unité lexicale dont le nom n'est pas celui d'un terminal du niveau syntaxique.

Cette possibilité est utilisée de façon intensive dans la description du niveau lexical de LECL lui-même (voir en annexe B.2).

Afin de déterminer statiquement (à la construction) si un terminal du langage est un mot clé, LECL recherche s'il existe au moins une expression régulière qui décrit le terminal considéré (le terminal appartient-il au langage défini par l'expression régulière ?). Or les expressions régulières peuvent contenir des actions et des prédicats utilisateur qu'il est bien entendu impossible d'exécuter à la construction. La possibilité de manipuler des variables globales (dont les valeurs peuvent être utilisées d'une unité lexicale à une autre) spécifiant par exemple des contextes de reconnaissance, interdit également la connaissance statique du résultat de l'exécution de prédicats système. Afin de ne pas "oublier" des possibilités de reconnaissance de mots clés, LECL suppose, pour cette reconnaissance des mots clés à la construction, que tous les prédicats dynamiques retournent `VRAI`. De ce fait, LECL est susceptible de décrèter que telle unité lexicale reconnaît tel mot clé, alors qu'il n'en est rien. Généralement, cet abus n'est pas dommageable.

Pour des raisons analogues il n'est pas en général possible pour LECL de connaître à la construction la forme interne d'une chaîne résultant de la reconnaissance d'une chaîne source. LECL est donc susceptible de "rater" la reconnaissance de certains mots clés.

Afin de pallier cette faiblesse et de permettre un contrôle plus fin, l'utilisateur a la possibilité de modifier le comportement précédent en indiquant explicitement si telle ou telle unité lexicale reconnait tel ou tel mot clé. Pour ce faire il est possible d'insérer une clause mot clé après la définition d'une unité lexicale. Cette clause spécifie le comportement de cette unité lexicale vis à vis des mots clés.

Cette clause, si elle est présente, doit suivre

- le "`;`" marquant la fin d'une expression régulière définissant une unité lexicale
- ou le "`;`" marquant la fin d'une spécification de "post-action"
- ou le "`;`" marquant la fin d'une clause contexte.

Elle peut avoir les formes suivantes :

- `NOT KEYWORD [ALL] ;` indique qu'en aucun cas, l'unité lexicale à laquelle elle est rattachée ne peut reconnaître de mot clé.
- `KEYWORD [ALL] ;` indique que l'unité lexicale à laquelle elle est rattachée reconnait tous les mots clés.
- `KEYWORD a, b, ... ;` indique que l'unité lexicale à laquelle elle est rattachée ne reconnait que les mots clés de la liste (`a`, `b`, ...).
- `KEYWORD ALL BUT a, b, ... ;` indique que l'unité lexicale à laquelle elle est rattachée ne reconnait que les mots clés qui n'appartiennent pas à la liste (`a`, `b`, ...).

Remarque : Pour des raisons de compatibilité avec les versions précédentes `NOT` et `KEYWORD` sont des mots clés non réservés de LECL.

Remarque : Les unités lexicales `Comments` et `Include` ne permettent pas de définir des mots clés.

4.1.4 Les synonymes

Si cette rubrique est présente, elle est introduite par le mot clé `SYNONYMS`. Elle permet d'associer une liste de synonymes à chaque terminal du niveau syntaxique.

Deux synonymes sont deux symboles qui vont jouer exactement le même rôle du point de vue syntaxique (ils auront le même code interne), mais qui ont des représentations différentes au niveau lexical.

La spécification de synonymes, utilisée essentiellement pour les mots clés, permet de diminuer le nombre des symboles terminaux de la grammaire syntaxique, ce qui diminue en conséquence la taille de l'automate à pile engendré par le constructeur syntaxique.

FIG. 4.2 – Exemple de définition de synonymes.

```
Synonyms
   DECLARE   = DECL, DCL ;
   PROCEDURE = PROC ;
   POINTER   = PTR ;
```

"`DECLARE`" est un terminal du niveau syntaxique, tandis que "`DECL`" et "`DCL`" n'en sont pas. Cette définition signifie que trouver "`DECL`" ou "`DCL`" dans un texte est équivalent à y trouver "`DECLARE`".

4.1.5 La spécification de représentation

Cette dernière rubrique, si elle est présente, permet de construire un analyseur lexicographique pour une machine différente de la machine hôte. On peut indiquer la représentation interne des caractères utilisée par la machine cible,

ainsi que les longueurs de l'octet—*byte*—et du mot—*word*—de cette machine cible.

La syntaxe est la suivante—n et m étant deux entiers positifs :

`For Internal Code Use` *liste_de_composants* `;`
`For Byte Use` n `Bits ;`
`For Word Use` m `Bits ;`

Les *composants* sont séparés par des virgules et indiquent, pour chaque index (en partant de 0), si le nombre associé correspond à une représentation interne d'un caractère ou non, et dans l'affirmative, précisent ce caractère :

– "`UNUSED`" ou "n `UNUSED`" : un nombre ou n nombres consécutifs ne sont pas des représentations internes de caractères ;
– un code octal ou une chaine de caractères : fournissent une ou plusieurs correspondances "caractère ⇌ représentation interne".

FIG. 4.3 – Exemple de spécification de représentation.

```
For Internal Code Use 10 Unused, "01", Unused, #012 ;
```

Les représentations internes sont : *10* pour le caractère "0", *11* pour le caractère "1" et *13* pour le caractère ASCII de code 10—"012" en octal. Dans la machine cible, les codes internes *0* à *9* et *12* correspondent à des caractères illégaux.

4.1.6 Les expressions régulières

Le vocabulaire des expressions régulières est formé par les noms de classes (prédéfinies, nommées ou anonymes) , d'abréviations, d'actions et de prédicats.

Une expression régulière est une liste d'alternatives séparées par des "`|`" (ou).

Une alternative est définie par les règles suivantes :

<*alternative*> `: :=` <*alternative*> <*facteur*>
`|` <*facteur*>

<*facteur*> `: :=` `(` <*expression régulière*> `)`
`|` `{` <*expression régulière*> `}`
`|` `{` <*expression régulière*> `}*`
`|` `(` <*expression régulière*> `)*`
`|` `(` <*expression régulière*> `)+`
`|` `{` <*expression régulière*> `}+`
`|` `[` <*expression régulière*> `]`
`|` <*primaire*>

Et <*primaire*> est un élément du vocabulaire.

Les parenthèses "`(`" et "`)`" ont la signification usuelle.

Les couples "`{`"–"`}`", "`{`"–"`}*`" et "`(`"–"`)*`" sont utilisés pour symboliser l'opérateur de fermeture transitive réflexive.

Les couples "`(`"–"`)+`" et "`{`"–"`}+`" sont utilisés pour symboliser l'opérateur de fermeture transitive.

Le couple "`[`"–"`]`" indique que l'expression régulière est facultative.

FIG. 4.4 – Exemple d'expression régulière.

Les identificateurs du langage d'entrée LECL peuvent être définis par l'expression régulière :

```
LETTER {["_"] (LETTER|DIGIT)}
```

4.1.6.1 L'opérateur "-"

Certains caractères reconnus au niveau lexical n'ont aucune utilité pour l'analyse syntaxique. Par exemple, dans une chaîne de caractères, les guillemets de début et de fin peuvent être éliminés, en ne conservant que les caractères utiles.

L'utilisateur a la possibilité de spécifier si un groupe de caractères doit être ou non conservé par l'analyseur lexical, en faisant précéder le *facteur* qui correspond à ce groupe de l'opérateur unaire "-". Cette suppression doit être cohérente, c'est-à-dire que deux occurrences du même caractère atteintes simultanément lors d'une analyse lexicale ne peuvent à la fois être conservées et supprimées.

FIG. 4.5 – Exemples d'utilisation de l'opérateur "-".

Les deux expressions régulières suivantes sont *a priori* licites :

```
-QUOTE {tout_sauf_quote | -QUOTE QUOTE} -QUOTE

-({SP|EOL}+ | "-" "-" {tout_sauf_EOL} EOL)
```

En revanche, la description suivante est erronée :

-QUOTE$_1$ {tout_sauf_quote | QUOTE$_2$ -QUOTE$_3$} -QUOTE$_4$

En effet, lors de la reconnaissance d'une chaîne, les occurrences 2 et 4 de `QUOTE` sont atteintes en parallèles. S'agit-il du guillemet de fin ou d'un guillemet intérieur ? Seule la connaissance du symbole suivant va permettre d'en décider. Il est donc erroné dans ce cas de spécifier simultanément une suppression (occurrence 4) et une mémorisation (occurrence 2).

Attention : L'opérateur binaire "-" utilisé dans les classes (différence ensembliste) ne peut pas être utilisé dans les expressions régulières.

4.1.6.2 L'opérateur "^"

C'est un opérateur unaire dont l'opérande est une classe et dont le résultat est le complémentaire de la classe opérande par rapport à `ANY`.

4.1.6.3 Les actions

Afin de permettre à l'utilisateur d'exécuter, au cours de l'analyse lexicale, des traitements qui se spécifient mal—ou ne peuvent pas se spécifier—par le formalisme des expressions régulières, il a été introduit des *actions* dans la description lexicale. Ces actions peuvent être considérées comme le pendant des "actions sémantiques" du niveau syntaxique.

FIG. 4.6 – Exemples d'utilisation de l'opérateur “^”.

L'expression régulière définissant une chaîne peut s'écrire :

```
-QUOTE {^QUOTE | -QUOTE QUOTE} -QUOTE
```

L'écriture

```
^"<"&Is_First_Col
```

(voir en 4.1.6.4) a la signification “`(^"<")&Is_First_Col`”, c'est-à-dire ‘caractère quelconque différent de “<” qui se trouve en première colonne’.

Tout nom d'action commence par le caractère “`@`”. Certaines actions d'intérêt général sont prédéfinies par le système LECL. Ces actions pourraient pour la plupart être écrites très simplement par l'utilisateur ; leur existence se justifie toutefois, non seulement par des considérations d'efficacité, mais encore par l'amélioration qu'elles peuvent apporter à la clarté de la spécification : elles portent en effet un nom significatif de leur fonction, alors que les actions de l'utilisateur ne peuvent être référencées que par un numéro.

Lors de l'exécution d'une action, que ce soit une action prédéfinie ou une action de l'utilisateur, le caractère suivant dans le texte source a toujours été lu, mais n'a pas encore été (pleinement) exploité ; il est donc accessible, si besoin est, par l'intermédiaire du module de gestion du texte source, dans la structure “`sxsrcmngr`” (voir la documentation en ligne—`man sxsrcmngr`).

Actions prédéfinies. Les actions prédéfinies par le système sont les suivantes :

`@LOWER_CASE` : Passe en minuscule tous les caractères qui ont été conservés depuis le début de la reconnaissance de l'unité lexicale courante.

`@UPPER_CASE` : Passe en majuscule tous les caractères qui ont été conservés depuis le début de la reconnaissance de l'unité lexicale courante.

`@LOWER_TO_UPPER` : Passe en majuscule le dernier caractère conservé.

`@UPPER_TO_LOWER` : Passe en minuscule le dernier caractère conservé.

`@PUT (`*char*`)` : Insère le caractère *char* dans la chaîne conservée.

`@MARK` : Mémorise la position courante dans la chaîne conservée—voir les actions `@ERASE` et `@RELEASE`.

`@ERASE` : Supprime tous les caractères qui ont pu être conservés depuis le `@MARK` précédent (dans la même unité lexicale) s'il existe ou sinon depuis le début de la reconnaissance de l'unité lexicale courante.

`@RELEASE` : *Repousse* dans le source les caractères mémorisés depuis le `@MARK` précédent (dans la même unité lexicale) s'il existe ou sinon depuis le début de l'unité lexicale courante.

Les actions suivantes permettent la gestion de compteurs (numérotés à partir de 0). Ces compteurs peuvent être testés par l'intermédiaire des prédicats prédéfinis `&Is_Set` et `&Is_Reset`—voir plus loin. Ils sont également manipulables par des actions *vel* des prédicats de l'utilisateur (voir la documentation

en ligne—`man sxscanner`). Ces compteurs sont initialisés à zéro au début de l'analyse de tout texte source.

`@RESET` (n) : Mise à zéro du compteur numéro n.

`@SET` (n) : Mise à un du compteur numéro n.

`@INCR` (n) : Le compteur numéro n est incrémenté de 1.

`@DECR` (n) : Le compteur numéro n est décrémenté de 1.

Actions de l'utilisateur. Les actions utilisateur sont définies par l'apparition de leur nom dans une expression régulière (caractère "`@`" suivi d'un nombre entier positif ou nul). A chacune de ces actions doit correspondre une portion de programme, écrite par l'utilisateur, réalisant la fonction désirée. Ces actions sont exécutées au cours de l'analyse lexicale d'un texte source.

Exemple : En Ada, la casse, majuscule ou minuscule, des lettres composant un identificateur est indifférente. Le même nom peut donc s'écrire tantôt en majuscules, tantôt en minuscules ou en mélangeant les deux types de capitalisation. Il est possible de mémoriser les identificateurs sous une forme normale, ne comportant par exemple que des lettres majuscules ; la transformation d'un identificateur quelconque vers cette forme normale (passage des minuscules en majuscules) peut être programmée en utilisant une action système. Cette action peut soit passer une seule lettre de minuscule en majuscule (action `@LOWER_TO_UPPER`) soit transformer l'ensemble de l'identificateur (action `@UPPER`). Suivant le cas on obtient la description

```
LETTER @LOWER_TO_UPPER {["_"] (LETTER @LOWER_TO_UPPER | DIGIT)}
```

si la transformation se fait lettre après lettre, ou

```
LETTER {["_"] (LETTER | DIGIT} @UPPER_CASE
```

si la transformation s'effectue globalement.

Un autre exemple : Certains langages de programmation spécifient que seuls un certain nombre de caractères sont significatifs dans les identificateurs, ou même limitent leur taille. La troncature ou la vérification nécessaires peuvent aisément se réaliser à l'aide du mécanisme des actions.

Une des originalités de LECL est d'utiliser les contextes droits (sur une longueur éventuellement non bornée) pour décider de l'action à exécuter, de la même façon que le contexte droit sert à la reconnaissance d'une expression régulière (voir en 4.1.7.1).

Exemple : La définition des deux unités lexicales

```
UL1      = "a" @1 {"b"} "c" ;
UL2      = "a" @2 {"b"} "d" ;
```

est licite. C'est la vue des caractères "`c`" ou "`d`" après un nombre non borné de "`b`" qui décidera de l'exécution de l'action "`@1`" ou "`@2`".

Remarque : Bien entendu, certaines des variables utilisées par l'analyseur lexical sont accessibles (et même modifiables...) par l'utilisateur, en particulier depuis le programme codant ses actions et prédicats (voir en 4.2 et consulter la documentation en ligne—`man sxscanner`).

Post-action. Il est également possible de spécifier une action utilisateur. Cette action, si elle existe, doit suivre

- le "`;`" marquant la fin de l'expression régulière à laquelle elle s'applique,
- ou le "`;`" marquant la fin de la clause `Priority` (voir en 4.1.7.3), si elle est présente,
- ou le "`;`" marquant la fin de la clause NOT" KEYWORD",
- ou le "`;`" marquant la fin de la clause contexte.

Une telle action est appelée *post-action* : lors de l'analyse lexicale, après reconnaissance de l'expression régulière à laquelle elle est attachée, cette action sera exécutée immédiatement avant le retour à l'analyseur syntaxique. L'utilisateur a donc la possibilité d'y modifier les informations calculées par l'analyseur lexical (code interne du terminal reconnu, informations stockées dans la "string-table", *et cetera*).

4.1.6.4 Les prédicats

Jusqu'à présent, nous avons supposé que la signification d'un caractère donné ne dépendait que de ce caractère. Il existe cependant un certain nombre de situations où cette signification dépend également de l'environnement : positionnement du caractère à un numéro de colonne donné, occurrence dans une portion de texte donnée, *et cetera*.

En général, ce genre de situation ne peut que difficilement se décrire de façon purement syntaxique. Afin de résoudre ce problème, LECL permet d'influer sur l'analyse lexicale, par l'intermédiaire de *prédicats*.

Il est possible, dans une expression régulière quelconque, d'associer à une occurrence d'une classe de caractères quelconque un prédicat système ou utilisateur. Un nom de prédicat commence par le caractère "`&`". Lors de l'analyse d'un texte source, un caractère t ne peut être reconnu par le couple *classe–prédicat* que si t appartient à *classe* et si *prédicat* est `vrai`.

Dans une expression régulière, toute occurrence d'un nom de classe n'ayant pas de prédicat associé peut être considérée comme étant associée à un prédicat toujours `vrai`.

À chaque prédicat—autre qu'un prédicat *statique*, voir ci-dessous—doit correspondre une fonction à valeur booléenne (définie par le système pour ce qui concerne les prédicats prédéfinis, ou écrite par l'utilisateur) ; ces fonctions sont exécutées au cours de l'analyse lexicale d'un texte source.

Les prédicats système

Statiques : Ces prédicats, contrairement aux autres, sont évalués à la construction et permettent de simplifier l'écriture des expressions régulières ; il s'agit des prédicats `&TRUE` et `&FALSE`.

Le prédicat `&TRUE` a la signification suivante :

> si, au cours de la construction de l'automate, LECL se trouve en plusieurs endroits différents dans le graphe des états (non-déterminisme) et qu'un de ces endroits au moins est une classe associée au prédicat `&TRUE`, alors le constructeur ne conserve que les endroits ainsi repérés, les autres étant considérés comme des impasses.

Exemples :

– Les expressions régulières de la figure 4.5, reconnaissant les chaînes de caractères et les commentaires, peuvent s'écrire :

```
-QUOTE {ANY |-QUOTE&TRUE QUOTE} -QUOTE&TRUE

-({SP|EOL}+ | "-" "-" {ANY} EOL&TRUE )
```

La classe composée `ANY` contient le caractère guillemet (`QUOTE` appartient à `ANY`) ; lorsqu'un guillemet est reconnu, l'on se trouve en trois endroits différents dans l'expression régulière :

```
-QUOTE {ANY | -QUOTE&TRUE QUOTE} -QUOTE&TRUE
        ↑      ↑                  ↑
```

Selon la règle précédente, seuls les deux derniers emplacements sont conservés. Le guillemet est donc reconnu par `QUOTE` et non par `ANY`.

– Sans utiliser de prédicat `&TRUE`, les commentaires PL/1 peuvent être reconnus par l'expression régulière

```
"/" "*" {^"*" | {"*"}+ ^"*/"} {"*"}+ "/"
```

En utilisant le prédicat `&TRUE`, on obtient l'expression ci-dessous, notablement plus simple et plus lisible !

```
"/" "*" {ANY} "*" "/"&TRUE
```

Dynamiques :

`&IS_FIRST_COL` : Vrai si et seulement si les caractères de la classe courante sont en première colonne.

`&IS_LAST_COL` : Vrai si et seulement si les caractères de la classe courante sont sur la dernière colonne d'une ligne.

`&IS_RESET` (n) : Vrai si et seulement si le compteur numéro n est nul.

`&IS_SET` (n) : Vrai si et seulement si le compteur numéro n est non nul.

Les prédicats utilisateur. Ils désignent, soit une fonction booléenne de l'utilisateur (caractère "`&`" suivi d'un nombre entier positif ou nul), soit un *nom d'expression booléenne* (caractère "`&`" suivi d'un identificateur). Dans ce deuxième cas, l'*expression booléenne* a dû être définie au préalable dans le paragraphe "`ABBREVIATIONS`" de la façon suivante :

<*nom de l'expression*> `::=` <*expression bool*>

où

- les opérateurs de <*expression bool*> sont :
  - la concaténation par juxtaposition (opération *et*),
  - l'opérateur "`|`" (opération *ou*),
  - l'opérateur "`^`" (opération de complémentation),

  avec les précédences usuelles ;
- les opérandes sont des prédicats ; chaque prédicat peut être soit un prédicat système *dynamique*, soit un prédicat utilisateur, soit un nom d'expression de prédicats défini préalablement.

Exemples

1. La figure 4.7 spécifie qu'un commentaire commence par le caractère "`*`" vérifiant la condition : être en première colonne et variable numéro 2 nulle, ou ne pas être en première colonne et variable numéro 2 non nulle.

FIG. 4.7 – Exemple de définition d'expression de prédicats.

```
Abbreviations
   &COND = &Is_First_Col &Is_Reset(2) |
           ^&Is_First_Col &Is_Set(2);
Tokens
   Comments = "*"&COND {^EOL} EOL;
```

---

2. Dans les langages *formattés*, la signification d'une unité lexicale peut dépendre de l'emplacement de son occurrence dans une ligne. Prenons comme exemple de spécification celle de BNF, dans laquelle on impose que les non-terminaux de la partie gauche d'une règle commencent en colonne 1. Ces symboles (`%LHS_NON_TERMINAL`) et les non-terminaux de la partie droite (`%NON_TERMINAL`) peuvent se décrire comme en figure 4.8, où le prédicat `&IS_FIRST_COL` répond `vrai` si et seulement si le caractère auquel il est associé (ici "<") se trouve en colonne 1 ; dans le cas contraire, on considère que le "<" est le débutant de `%NON_TERMINAL`.

FIG. 4.8 – Extrait de la spécification lexicale de BNF.

```
%LHS_NON_TERMINAL = "<"&IS_FIRST_COL {^EOL} ">"&TRUE ;
%NON_TERMINAL     = "<" {^EOL} ">"&TRUE ;
```

3. Le mécanisme des actions et des prédicats augmente la puissance théorique de description des expressions régulières ; il est possible de décrire des langages non réguliers, par exemple des commentaires imbriqués.
   Supposons que les caractères "{" et "}" marquent respectivement le début et la fin d'un commentaire et qu'il soit possible de trouver un commentaire à l'intérieur d'un autre commentaire—ce type de commentaire ne se rencontre à l'heure actuelle que très rarement dans les langages, malheureusement. Une description possible est donnée en figure 4.9.

FIG. 4.9 – Exemple de spécification des commentaires de *OLGA*.

```
Comments = "{" @Reset (0)
               {
                ^"{}"                   |
                "{" @Incr (0)           |
                "}"&Is_Set (0) @Decr (0)
               }
           "}" ;
```

Remarque : Plus généralement, il est possible d'utiliser LECL pour décrire des langages non réguliers en simulant l'automate à pile correspondant par des appels récursifs de l'analyseur lexical, dans un style qui s'apparente à la descente récursive dans les analyseurs syntaxiques de la classe LL. À chaque notion du langage (non-terminal) on fait correspondre une description (langage) et chaque fois qu'à un niveau donné on reconnait, grâce à un préfixe, la présence d'une

notion, l'analyseur lexical de cette notion est appelé par l'intermédiaire d'une action utilisateur. Lorsque cette notion a été traitée, une autre action utilisateur permet le retour au niveau appelant.

Par exemple, la spécification lexicale précédente des commentaires OLGA, peut également se décrire comme en annexe F.2.

4.1.6.5 Les *includes*

L'utilisateur a la possibilité de définir des *commandes d'inclusion*, introduites par le mot clé `INCLUDE` dans la rubrique `TOKENS`. Cette définition doit comporter une expression régulière qui reconnaît la commande et deux actions utilisateur (dont une *post-action*). La première action doit assurer le changement de fichier source (en fonction des informations extraites de la commande) alors que la post-action doit assurer le retour normal à la lecture du fichier courant. Le module de librairie `sxincl_mngr` (*include manager*) peut être utilisé (voir la documentation en ligne—`man sxincl_mngr`).

Exemple : Soit un langage où la commande `include` est introduite par le caractère "`#`" et se termine sur la première fin de ligne. Les caractères non blancs sont censés représenter le *chemin* (*pathname*) du fichier à considérer. La description lexicale correspondante peut donc être :

```
Include = -"#" -{SP | HT} {^" \t\n"}+ -{SP | HT} -EOL @1 ; @2 ;
```

On trouvera en annexe F.3 le texte du programme codant les actions `@1` et `@2`.

4.1.7 Les conflits

Les langages décrits par les expressions régulières de la rubrique `TOKENS` doivent être disjoints deux à deux, c'est-à-dire que lors d'une analyse lexicale, toute séquence de caractères ne doit pouvoir être reconnue que par une seule expression régulière. Sinon on a un conflit potentiel qui est dit *Reduce/Reduce*.

De même une chaîne reconnue par une expression régulière donnée ne peut être le préfixe d'un autre chaîne, sinon on a un conflit potentiel qui est dit *Shift/Reduce*.[1]

Afin de résoudre de tels conflits, LECL met à la disposition de l'utilisateur deux mécanismes : les contextes qui font l'objet de 4.1.7.1 et les priorités qui sont traitées en 4.1.7.3.

4.1.7.1 Les contextes

La grammaire du niveau syntaxique définit les séquences valides d'unités lexicales mais ne précise pas la façon dont elles vont être séparées les unes des autres.

Considérons par exemple un langage où un nombre peut suivre un identificateur ; la chaîne "`A1`" peut être considérée soit comme un seul identificateur de texte `A1`, soit comme la séquence identificateur `A`–nombre `1`.

Les séquences valides d'unités lexicales (déduites de la grammaire syntaxique) permettent également, dans certains cas, de lever des ambigüités. Soit un

[1] Les conflits *Shift/Action* n'existent pas—ce ne sont que des mirages !

langage où "`*`" et "`**`" sont des opérateurs valides. La vue d'une étoile ne suffit pas pour décider si l'on a reconnu l'opérateur `*` ou si l'on est dans l'opérateur `**`.

En revanche, si l'on suppose que l'opérateur `*` ne peut pas être suivi d'une unité lexicale commençant par une étoile, la connaissance d'un caractère en avance (*look-ahead* d'un caractère) permet de résoudre ce conflit : si ce caractère en avance est une étoile, on est dans la reconnaissance de `**`, sinon on vient de reconnaître `*`.

Les *clauses contextes* permettent de préciser les *suivants immédiats* possibles des unités lexicales et des commentaires.

On appelle *suivants immédiats* l'ensemble des unités lexicales pouvant suivre une unité lexicale donnée, conformément à la grammaire du niveau syntaxique et aux clauses contextes.

Attention : cette notion est définie en terme d'unités lexicales et non en terme de terminaux du niveau syntaxique.

Exemple : Supposons que le mot clé `THEN` puisse, d'après la grammaire, suivre un nombre entier (terminal générique `%integer`) et que ce mot clé soit reconnu par l'unité lexicale `%identifier` ; alors `%identifier` est un élément des suivants immédiats de `%integer`.

En l'absence de clause contexte, les suivants immédiats sont :
- pour un commentaire : toutes les unités lexicales ainsi que les commentaires ;
- pour une unité lexicale : les commentaires plus les unités lexicales reconnaissant les terminaux qui peuvent suivre, d'après la grammaire, n'importe quel terminal reconnu par cette unité lexicale.

La clause contexte, si elle existe, doit suivre
- le "`;`" marquant la fin de l'expression régulière à laquelle elle s'applique,
- ou le "`;`" marquant la fin de la clause `Priority` (voir en 4.1.7.3), si elle est présente,
- ou le "`;`" marquant la fin de la spécification d'une "post-action" (voir en 4.1.6.3),
- ou le "`;`" marquant la fin de la clause NOT" KEYWORD",
- ou le "`;`" marquant la fin d'une autre clause contexte.

Cette clause contexte se termine par un point-virgule et peut s'écrire de trois manières différentes :

1. [`Unbounded`] `Context` *liste_de_suivants* ;
2. [`Unbounded`] `Context All` ;
3. [`Unbounded`] `Context All But` *liste_de_suivants* ;

Dans le premier cas, on indique explicitement la liste des suivants valides. Dans le deuxième cas, on indique que les suivants valides sont ceux déduits de la grammaire syntaxique, sans modification. Le troisième cas est analogue au précédent, excepté que *liste_de_suivants* est supprimée des suivants valides.

Les suivants (séparés par des virgules) peuvent être :
- un nom d'unité lexicale (identificateur ou chaîne de caractères) ;
- un nom d'un composant d'union (voir ci-dessous).

Ces contextes peuvent être utilisés de deux façons pour résoudre des conflits :

1. si le mot clé `Unbounded` est absent, on est en mode *1_look_ahead* ; LECL essaie alors de résoudre les conflits à la vue d'un seul caractère en avance ;

2. si le mot clé `Unbounded` est présent, LECL peut utiliser un nombre non borné de caractères en avance. Une résolution `Unbounded` sur une unité lexicale, dont le contexte est directement donné par la grammaire du niveau syntaxique et non modifié par une clause `Context` explicite, doit être introduite par `Unbounded Context All;`.

4.1.7.2 Les unions

Afin d'augmenter la précision de la description des séquences valides d'unités lexicales, il est possible de définir une unité lexicale comme étant décrite par plusieurs expressions régulières. Cette possibilité, appelée *union*, est introduite par le mot clé `UNION` et se termine par le mot clé `END`—voir l'exemple de la figure 4.10. On référence les membres d'une union par une notation pointée :

nom-de-l'unité. *nom-du-composant*.

Il peut y avoir des clauses mot clé (voir en 4.1.3.1), post-action (voir en 4.1.6.3), contexte (voir en 4.1.7.1) et priorité (voir en 4.1.7.3) associées à chaque composant de l'union. Par défaut chaque composant hérite les clauses (s'il y en a) de l'unité lexicale, qui sont spécifiées après le `End ;` marquant la fin de l'union. Pour une clause donnée, cet héritage est inhibé si pour un composant d'une union cette clause a été redéfinie au niveau de ce composant.

FIG. 4.10 – Exemple d'utilisation de l'union.

Considérons la définition suivante—extraite de la spécification du langage Ada :

```
Comments =  Union
               blanks : {SP|HT|EOL}+ ;
               ada    : "-" "-" {^EOL} EOL ;
            End ;
```

Cette définition, équivalente en ce qui concerne la reconnaissance des commentaires à

```
Comments = {SP|HT|EOL}+ | "-" "-" {^EOL} EOL ;
```

permet par exemple d'interdire à la deuxième forme des commentaires de suivre l'opérateur *moins*, en le définissant par

```
"-" = "-" ; Context All But Comments.ada;
```

ce qui signifie que la présence des caractères "`---`" en début d'une unité lexicale n'a pas la signification *opérateur moins suivi d'un début de commentaire*.

4.1.7.3 Les priorités

Cette clause, introduite par le mot clé `PRIORITY`, doit suivre, si elle est présente

- le ";" marquant la fin d'une expression régulière
- ou le ";" marquant la fin d'une spécification de "post-action"
- ou le ";" marquant la fin d'une clause contexte,
- ou le ";" marquant la fin d'une clause "NOT KEYWORD",
- ou le ";" marquant la fin d'une autre clause `PRIORITY`.

Elle est formée d'une liste d'au plus trois spécifications de priorité séparées par des "`,`" ; cette clause se termine par un "`;`".

Les possibilités de spécification de priorité sont les suivantes :

`Reduce>Reduce` : En cas de conflit (*1_look_ahead* ou *Unbounded*), la priorité est donnée à l'expression courante sur toute autre expression ne comportant pas cette clause et reconnaissant un (sous-)langage commun.

`Reduce>Shift` : En cas de conflit (*1_look_ahead* ou *Unbounded*), les chaînes reconnues par l'expression régulière courante (*Reduce*) ont priorité sur les préfixes stricts (*Shift*) de chaînes reconnues soit par cette expression régulière soit par d'autres.

`Shift>Reduce` : En cas de conflit (*1_look_ahead* ou *Unbounded*), toute chaîne reconnue par une expression (*Reduce*) comportant cette clause est abandonnée au profit des préfixes stricts (*Shift*) déjà reconnus, s'il y en a.

Les clauses priorité sont appliquées successivement dans l'ordre indiqué.

FIG. 4.11 – Exemples d'utilisation de clause `Priority`.

On définit les deux terminaux génériques suivants :

```
%IDENT   = LETTER {["_"] (LETTER|DIGIT)} ;
%CAR_CDR = "C" {"A" | "D"}+ "R";
           Priority Reduce>Reduce;
```

et l'on suppose que leurs contextes sont identiques.
La chaîne "`CADDR`" est reconnue par les deux expressions régulières, mais à cause de la clause de priorité sera considérée comme étant un `%CAR_CDR`.
Notons que la chaîne "`CADDR1`" sera considérée comme étant un `%IDENT`. En revanche, avec la description suivante :

```
%IDENT   = LETTER {["_"] (LETTER|DIGIT)};
%NOMBRE  = {DIGIT}+;
%CAR_CDR = "C" {"A" | "D"}+ "R";
           Priority Reduce>Reduce, Reduce>Shift;
```

cette même chaîne source "`CADDR1`" sera considérée comme étant une suite `%CAR_CDR` – `%NOMBRE` ("`CADDR`" et "`1`").

4.1.7.4 Principes de la résolution des conflits

En cas de conflit, LECL applique les règles suivantes :

- La présence d'actions dans une expression régulière ne doit pas *a priori* modifier le résultat de l'analyse.
- Les décisions se prennent sur les réductions.
- On détermine tout d'abord le mode de résolution du conflit : *1_look_ahead* ou *Unbounded* : une résolution est *Unbounded* si et seulement si toutes les unités lexicales intervenant dans le conflit ont une clause `Unbounded Context` ; dans le cas contraire, le mode est *1_look_ahead*.
- S'il y a conflit pour le mode trouvé, LECL applique les priorités utilisateur (s'il y en a).
- S'il reste des conflits, LECL applique alors les règles de résolution par défaut : il donne priorité à l'action (*Shift*, *Action*, *Reduce*) qui assure

la reconnaissance de la chaîne la plus longue à l'intérieur d'une même expression régulière. En cas d'égalité, il donne priorité à l'action rencontrée en premier dans un parcours gauche droit de la grammaire lexicale et produit un diagnostic.

Exemple Considérons le langage formé par une liste d'identificateurs ("%ID") et de nombres entiers ("%NB"), et qui a la définition lexicale suivante :

```
Tokens
        Comments = -{SP|EOL}+ ;
        %ID      = LETTER {LETTER | DIGIT} ;
        %NB      = {DIGIT}+ ;
```

LECL produit sur cette spécification les diagnostics de la figure 4.12. Comme il n'y a pas de clause `Unbounded Context`, tous ces conflits sont traités en mode *1_look_ahead*.

FIG. 4.12 – Exemple de diagnostics de LECL sur des conflits.

```
**** Warning : 1_look_ahead Shift/Reduce conflict in state 3 on LETTER between
               Shift :
                    %ID = LETTER {¸LETTER | DIGIT} ;
               Reduce :
                    %ID = LETTER {LETTER | DIGIT}¸ ;
                    detected on :
                         %ID = ¸LETTER {LETTER | DIGIT} ;
               Priority is given to Shift.
**** Warning : 1_look_ahead Shift/Reduce conflict in state 3 on DIGIT between
               Shift :
                    %ID = LETTER {LETTER | ¸DIGIT} ;
               Reduce :
                    %ID = LETTER {LETTER | DIGIT}¸ ;
                    detected on :
                         %NB = {¸DIGIT}+ ;
               Priority is given to Shift.
**** Warning : 1_look_ahead Shift/Reduce conflict in state 4 on DIGIT between
               Shift :
                    %NB = {¸DIGIT}+ ;
               Reduce :
                    %NB = {DIGIT}+¸ ;
                    detected on :
                         %NB = {¸DIGIT}+ ;
               Priority is given to Shift.
```

Le premier diagnostic signifie que si deux identificateurs se succèdent sans séparateur, il est impossible de les distinguer de l'occurrence d'un seul. Le traitement par défaut est dans ce cas de considérer que l'on a un seul identificateur (priorité est donnée à la chaîne la plus longue).

Le second et le troisième ont des significations analogues : identificateur suivi d'un nombre et nombre suivi d'un nombre.

Si l'on veut éviter ces messages, il est possible de spécifier, en utilisant les clauses contexte, qu'un identificateur ou un nombre ne peut pas suivre (immédiatement) un identificateur et qu'un nombre ne peut pas suivre un nombre :

```
%ID = LETTER {LETTER | DIGIT} ;
      Context All But %ID, %NB ;
%NB = {DIGIT}+ ;
      Context All But %NB ;
```

La résolution de ces conflits peut également être obtenue (sans diagnostic) en utilisant les clauses priorité :

```
%ID = LETTER {LETTER | DIGIT} ;
      Priority Shift>Reduce ;
%NB = {DIGIT}+ ;
      Priority Shift>Reduce ;
```

4.1.8 Exemples de définitions lexicales

On trouvera en annexe les spécifications Lecl du langage Pascal (annexe F.1) et des langages Bnf (A.2) et Lecl lui-même (B.2).

4.2 Mise en œuvre

Voir la documentation en ligne—`man lecl`. De plus, si l'utilisateur a inclu des actions *vel* des prédicats dans sa définition lexicale, il doit écrire un programme en langage C, codant ces actions et prédicats. Ce codage nécessite en général d'accéder à un certain nombre de variables manipulées par Syntax (se reporter à la documentation en ligne, concernant notamment `sxunix(3)`, `sxsrc_mngr(3)`, `sxscanner(3)` et `sxstr_mngr(3)`).

On trouvera des exemples de codage d'actions et de prédicats en annexes B.3 et A.3.

Chapitre 5

La sémantique dans SYNTAX

SYNTAX permet deux types distincts de traitement sémantique :

- soit en effectuant des *actions sémantiques* pendant l'analyse syntaxique : l'appel de l'action associée à une règle de la grammaire s'effectue après la reconnaissance de la partie droite de cette règle.
- soit en utilisant l'*arbre abstrait* que peut construire l'analyseur syntaxique et en appliquant sur cet arbre un programme d'évaluation d'*attributs sémantiques*.

Ces deux possibilités sont obtenues respectivement par utilisation des constructeurs SEMACT et SEMAT.

5.1 SEMACT : Actions sémantiques

Pour introduire des actions sémantiques dans la grammaire (contenue alors généralement dans un fichier de nom *nom-du-langage*`.bnf`), l'utilisateur doit ajouter, en fin de règle (après le "`;`"), un nombre entier. Ce nombre indique le numéro de l'action sémantique qui sera appelée par l'analyseur syntaxique au moment de la reconnaissance de la règle associée.

Dans le cas où une règle n'est pas suivie d'un numéro d'action le système ajoute automatiquement le nombre `0` à la fin de cette règle. Ainsi l'action numéro `0` sera appelée lors de la réduction de telles règles. (En général, cette action ne fait rien...)

L'utilisateur doit alors écrire un programme en langage C (ou dans un autre langage, après réalisation d'une interface) réalisant l'ensemble des actions sémantiques du langage. Ce programme doit correspondre à la structure donnée en annexe C.1.

5.2 SEMAT : Sémantique par arbre abstrait

5.2.1 Spécification et construction de l'arbre abstrait

La phase préliminaire de l'analyse sémantique par arbre abstrait est la production de cet arbre par le constructeur d'arbres abstraits (voir la documentation en ligne—`man sxatc`), au cours de l'analyse syntaxique.

Pour obtenir cette construction, il faut modifier la définition syntaxique (il est d'usage de nommer alors *nom-du-langage*.at le fichier correspondant) ; cette modification consiste en l'ajout de noms de nœuds, à la fin des règles (après le ";") ; ces noms sont des identificateurs "à la C" placés entre guillemets (""").

Lorsque l'analyseur syntaxique reconnait une règle, les racines des arbres correspondant aux non-terminaux et aux terminaux génériques de la partie droite deviennent les fils d'un nouveau nœud représentant le non-terminal de la partie gauche de la règle. Ce nouveau nœud porte le nom qui se trouve en fin de règle.

À un terminal générique correspond une feuille qui porte le même nom que ce terminal. Cependant, dans le cas où ce terminal générique se trouve dans une règle ne comportant pas d'autres terminaux génériques et aucun non-terminal et que cette règle est suivie par un nom de nœud, alors la feuille qui correspond au terminal générique porte ce nom.

Les règles de construction qui viennent d'être énoncées ne sont pas tout à fait systématiques. Il existe quelques exceptions importantes :

1. Lorsque l'analyseur syntaxique reconnait une règle ne comportant que des terminaux non génériques, ou une règle vide, il crée une feuille dont le nom est le nom de nœud associé—s'il est spécifié, sinon VOID.
2. Si le nom d'un non-terminal récursif à gauche se termine par "_LIST" (par "_RIGHT_LIST" s'il est récursif à droite), tous les éléments de la liste sont réunis sous un même nœud. Aucun nom de nœud ne doit suivre la (les) règle(s) récursive(s) définissant ce non-terminal. Si la (les) règle(s) définissant ce non-terminal et débutant (ou finissant) la récursion est suivie d'un nom de nœud, le nœud *liste* créé portera ce nom ; sinon il porte le nom du non-terminal de la partie gauche.
   Voir l'exemple de la figure 5.1.

FIG. 5.1 – Exemple de création d'arbre abstrait pour une liste.

Considérons les règles :

```
<OBJECT_LIST>   = <OBJECT_LIST> , <OBJECT> ;
<OBJECT_LIST>   = <OBJECT>                 ; "OBJ_S"
<OBJECT>        = .....                    ; "OBJ"
```

Si, dans un texte source, la liste d'objets contient trois objets, l'arbre syntaxique classique serait celui décrit en figure 5.2. En revanche, l'arbre abstrait engendré par le constructeur d'arbre est celui de la figure 5.3.

3. Il est possible de ne pas spécifier de nom de nœud pour une règle, si cette règle ne comporte qu'un seul non-terminal ou terminal générique en partie droite (éventuellement accompagné de terminaux non génériques). Lorsque l'analyseur syntaxique réduit une telle règle, il n'apporte aucune modification à l'arbre en cours de construction. Ces omissions permettent donc de condenser l'arbre abstrait engendré.
4. Enfin, en cas d'erreur syntaxique non corrigible (voir en 6.1.2), le constructeur d'arbres réunit tous les sous-arbres déjà construits et impliqués dans ce rattrapage global sous un nœud de nom ERROR.

Remarque : On peut donner le même nom de nœud à plusieurs règles

Fig. 5.2 – Exemple d'arbre syntaxique.

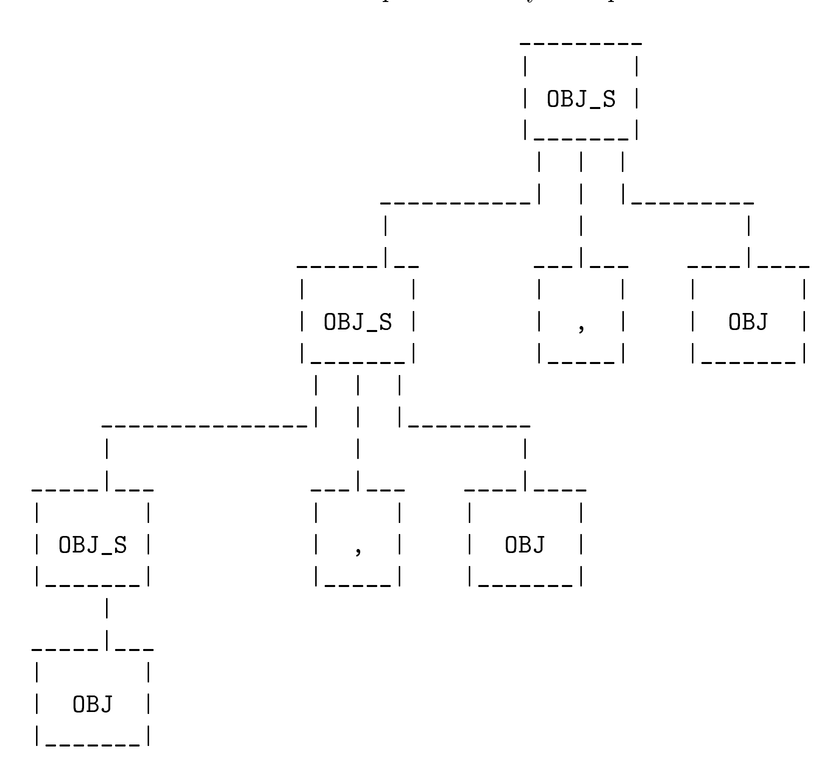
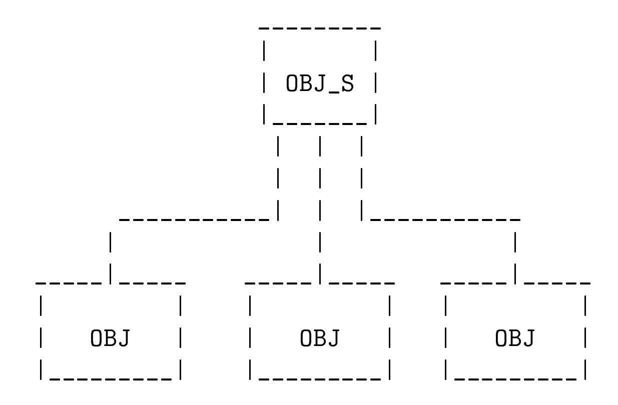


Fig. 5.3 – Exemple d'arbre abstrait.



différentes si elles ont la même arité (c'est à dire les nœuds représentant les non-terminaux de la partie gauche des règles ont le même nombre de fils).

Voir par exemple la grammaire décrivant le métalangage lexical en page 65.

Une fois la grammaire modifiée, on la soumet à l'entrée de SEMAT. En plus des sorties propres à BNF, SEMAT fournit, pour chaque nom de nœud et pour chacune des positions des fils, la liste de tous les noms de nœuds qui peuvent se trouver à cette position.

De plus, une trame de passe sémantique est engendrée, écrite en langage C. Cette trame peut être utilisée, après avoir été complétée *manuellement* par les calculs d'attributs, pour écrire l'analyseur sémantique.

5.2.2 Réalisation de l'analyseur sémantique

5.2.2.1 Évaluation des attributs sémantiques

Les attributs sémantiques sont associés aux nœuds de l'arbre abstrait. Une *passe sémantique* consiste en un parcours de cet arbre dans le sens "haut-bas, gauche-droite", en profitant des passages sur les nœuds pour évaluer les attributs. Plusieurs passes peuvent être réalisées de la sorte.

Chaque nœud est visité deux fois :

1. à l'entrée de son sous-arbre : lors de ce passage, les attributs *hérités* du nœud sont évalués. Cette visite sera dite "héritée".
2. à la sortie du sous-arbre : cette fois les attributs *synthétisés* sont évalués. Cette visite sera qualifiée de "synthétisée".

Depuis un nœud, il est possible d'accéder aux nœuds voisins (s'ils existent), à savoir : son père, ses frères et ses fils. Il est donc possible, en un nœud donné, de calculer un attribut de ce nœud en fonction d'attributs déjà calculés des nœuds voisins.

5.2.2.2 Les outils

Les outils nécessaires à la réalisation de l'analyse sémantique sont principalement la structure des nœuds et les pointeurs qui permettent d'y accéder, ainsi que les procédures de parcours de l'arbre abstrait (voir la documentation en ligne sur `sxatc(3)` et `sxat_mngr(3)`).

Tous les nœuds de l'arbre ont la même structure, et possèdent donc tous les attributs utilisés.

La macro "`VISITED`" s'étend en un pointeur vers le nœud *visité*—celui pour lequel on doit calculer des attributs.

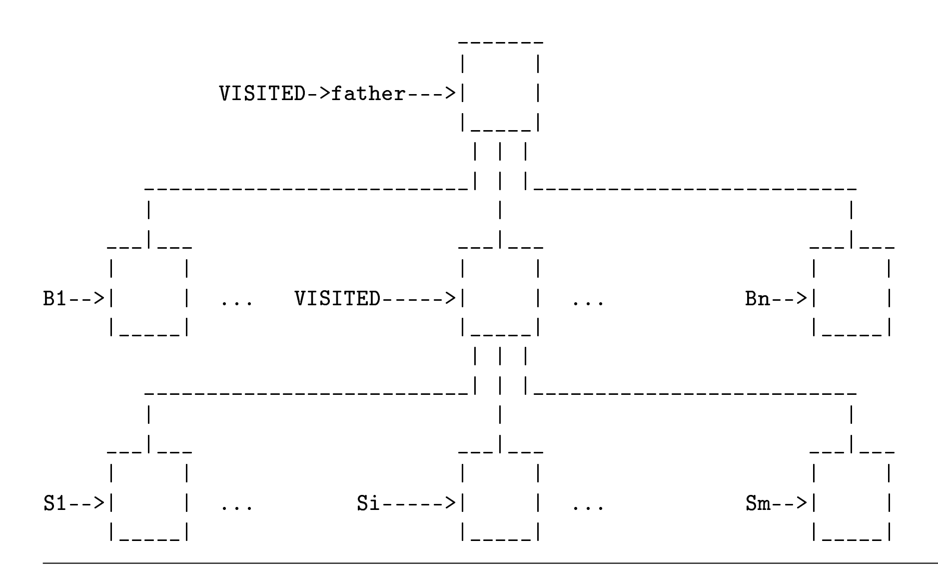
Les relations suivantes, qui font référence à la figure 5.4, sont vérifiées :

B_1 = `sxbrother (VISITED, 1)`
`VISITED` = `sxbrother (VISITED, VISITED->position)`
B_n = `sxbrother (VISITED,` n`)`
avec n = `VISITED->father->degree`

S_1 = `sxson (VISITED, 1)`
S_i = `sxson (VISITED,` i`)`
S_m = `sxson (VISITED,` m`)`
avec m = `VISITED->degree`

FIG. 5.4 – Accès à l'arbre abstrait.

Le sous-arbre *entourant* le nœud visité a la structure suivante :



Quelques *accélérateurs* peuvent être utilisés dans certains cas, pour améliorer la vitesse d'accès à un nœud particulier :

- On peut accéder au frère de droite (s'il existe) par `VISITED->brother` plutôt que par `sxbrother (VISITED, VISITED->position + 1)`.
- Si l'on est en visite héritée, le frère de gauche (s'il existe) peut être accédé par `LEFT` plutôt que par `sxbrother (VISITED, VISITED->position - 1)`.
- En visite synthétisée, le fils le plus à droite peut être accédé par `LAST_ELEM` plutôt que par `sxson (VISITED, VISITED->degree)`.

5.2.2.3 Mise en œuvre

L'évaluation des attributs est réalisée par des instructions qui sont à insérer dans la trame engendrée par SEMAT. Cette trame comporte essentiellement la définition de deux procédures :

- *nom-du-langage*`_pi` pour les attributs hérités,
- *nom-du-langage*`_pd` pour les attributs synthétisés.

Ces deux procédures sont constituées principalement par des instructions `switch`, le choix étant fait sur :

- le nom du nœud père, puis la position du fils visité, pour la passe héritée ;
- le nom du nœud visité pour la passe synthétisée.

L'entête d'une telle trame est constituée par les déclarations de constantes représentant les noms de nœud. Chaque nom de constante est formé par le nom du nœud suffixé par "`_n`".

Par exemple, au nœud de nom `OBJ` et de code interne 3 est associée la définition `#define OBJ_n 3`.

Les deux procédures "*nom-du-langage*`_pi`" et "*nom-du-langage*`_pd`" sont passées en paramètre à `sxsmp`, qui réalise une passe "haut-bas gauche-droite" sur l'arbre paramètre. Chaque fois qu'un nœud est visité en hérité, `sxsmp` appelle

*nom-du-langage*_pi, chaque fois qu'un nœud est visité en synthétisé, sxsmp appelle *nom-du-langage*_pd.

On trouvera en annexe C.2, à titre d'exemple, la passe sémantique produite automatiquement par SEMAT sur la grammaire syntaxique de BNF.

Remarques : Il est possible de spécifier que le prochain nœud visité sera "node" dans le *sens* "kind" (INHERITED ou DERIVED), en utilisant la macro "sxat_snv (kind, node)".

Le module MIX (voir la documentation en ligne—man mix) peut aider à gérer les versions successives d'une passe sémantique.

Chapitre 6

RECOR : Le traitement des erreurs

Pour tout langage "*nom-du-langage*", une spécification destinée au traitement des erreurs doit être décrite dans un fichier *nom-du-langage*`.recor`.

En général, ce fichier sera construit par modifications de `standard.recor`. Une modification de la spécification du traitement des erreurs ne nécessite que de réactiver le module RECOR, sans avoir à repasser les autres phases de la construction.

6.1 Traitement des erreurs syntaxiques

Le traitement des erreurs dans le système SYNTAX se décompose en deux parties :

1. une tentative de *correction locale* du programme source erroné ;
2. si la tentative précédente échoue, *récupération* de l'erreur détectée, au niveau global.

6.1.1 Correction locale

Supposons qu'une erreur de syntaxe soit détectée sur le symbole terminal a_1. Soit xa_0y le texte source où xa_0 est la portion de texte déjà analysée (a_0 est le symbole précédent a_1) et $y = a_1a_2a_3\ldots$ est la partie non encore analysée.

La méthode de correction locale s'énonce comme suit :

> On considère l'ensemble $\mathcal{V}$ de toutes les chaînes terminales de longueur n telles que, quelle que soit la chaîne t de $\mathcal{V}$, la chaîne xt soit (un préfixe de chaîne) syntaxiquement correcte.
>
> On choisit dans $\mathcal{V}$ une chaîne qui va remplacer le début de a_0y de telle façon que le texte source modifié *ressemble* au texte source initial. Cette *ressemblance* est décrite par l'utilisateur, qui spécifie une liste de modèles de priorité décroissante.

Considérons par exemple les trois modèles suivants numérotés de 1 à 3 :

1 :	0 X 1 2 3	priorité
2 :	0 X 2 3 4	décroissante
3 :	0 2 3 4	

Dans ces modèles, X a la signification "un symbole terminal quelconque du langage", et les nombres représentent les symboles terminaux du texte source : 0 pour a_0, 1 pour a_1, ...

Supposons que la chaîne " $a_0ba_2a_3a_4$ " soit dans $\mathcal{V}$. Cette chaîne satisfait le modèle numéro 2, ce qui est interprété de la façon suivante :

Le symbole a_1 est erroné mais peut être remplacé par le symbole b.

La chaîne d'entrée " $a_0a_1a_2a_3a_4$ " sera donc remplacée par " $a_0ba_2a_3a_4$ " à moins qu'il n'y ait dans $\mathcal{V}$ une chaîne qui satisfasse le premier modèle (" $a_0ca_1a_2a_3$ " par exemple) de plus forte priorité.

Les modèles peuvent évidemment être plus élaborés :

		Modèle	Interprétation
1	:	0 X 1 2 3	oubli d'un symbole
2	:	0 X 2 3 4	un symbole erroné
3	:	0 2 3 4	un symbole en trop
4	:	1 0 2 3	interversion de deux symboles
5	:	X 1 2 3 4	symbole précédent erroné
6	:	1 2 3 4	symbole précédent en trop

La longueur du plus long modèle spécifie la longueur n des chaînes de $\mathcal{V}$.

À la vue de ces modèles, on remarquera que l'on a la possibilité d'effectuer des corrections impliquant le symbole qui précède le symbole sur lequel l'erreur a été détectée.

Les modèles utilisés ne doivent pas être trop longs, pour ne pas augmenter de manière conséquente le temps d'exécution de la correction d'erreur. Il semble qu'un bon compromis soit atteint pour une valeur de n voisine de 4 ou 5.

SYNTAX comporte également un mécanisme assez général de correction des "fautes d'orthographe" sur les mots clés. Ce mécanisme peut être exploité lors de la tentative de correction d'une erreur mettant en cause un terminal générique dont la définition lexicale "reconnaît" un ou plusieurs mots clés. Il permet de corriger les fautes simples tombant dans l'une des quatre catégories suivantes :

– omission d'un caractère,
– ajout d'un caractère,
– remplacement d'un caractère,
– interversion de deux caractères.

Ce mécanisme sera mis en œuvre si l'on fournit des modèles dans lesquels un numéro de symbole terminal est remplacé par "S".

Exemple :

0 S 2	correction sur le symbole a_1
S 1	correction sur le symbole a_0

6.1.2 Récupération globale

Si aucune correction locale, gouvernée par les modèles fournis, ne peut produire une chaîne valide, l'on tente une action globale permettant la reprise de l'analyse.

Cette récupération globale utilise la liste de *terminaux de rattrapage* de "`Key_Terminals`" (voir en 6.3).

Lorsque la correction locale a échoué, le texte source est lu (sans analyse) jusqu'à rencontrer un des terminaux de rattrapage. L'analyseur regarde alors si la chaîne source de longueur p (p est donné par "`Validation_length`") commençant par ce terminal peut être un suivant possible de l'un des états de la pile d'analyse après transition sur un non-terminal :

- si oui, l'analyse repart de cet endroit et on dépile jusqu'à ce que l'on puisse *transiter* sur ce non-terminal à partir du sommet de pile. Tout se passe comme si la partie du texte non analysée, plus éventuellement une portion du texte déjà analysée, était considérée comme une chaîne dérivée de ce non-terminal ;
- si non, l'analyseur recherche dans le texte source le prochain terminal de rattrapage et il recommence.

Cette description du mécanisme de récupération globale suggère qu'il est préférable, pour permettre une *bonne* récupération d'erreurs, que les terminaux de rattrapage choisis soient placés, dans la grammaire, immédiatement après des non-terminaux représentant des *phrases de haut niveau* du langage—voir l'exemple de la figure 6.1.

FIG. 6.1 – Modification de la grammaire pour améliorer le rattrapage d'erreur.

Considérons les règles suivantes :

```
<LISTE DE DECL> = <LISTE DE DECL> <DECL> ;
<LISTE DE DECL> = <DECL>                 ;
<DECL>          = DCL %id #;             ;
```

Le "`;`" est *a priori* défini comme un terminal de rattrapage. Mais si une erreur se produit en début de déclaration, le "`;`" se trouvant en fin de déclaration ne pourra pas permettre de se récupérer car il n'est pas précédé d'un non-terminal. Pour permettre la récupération, il faut transformer la grammaire, par exemple de la façon suivante :

```
<LISTE DE DECL> = <LISTE DE DECL> <DECL> #; ;
<LISTE DE DECL> = <DECL> #;                 ;
<DECL>          = DCL %id                   ;
```

Remarque : Avant d'effectuer une récupération globale, l'analyseur regarde s'il n'existe qu'un seul symbole pouvant suivre la partie du texte source déjà analysée. Si c'est le cas, il l'insère automatiquement et il repart sur ce symbole. Il peut ainsi rajouter toute une séquence de symboles "obligatoires".

6.2 Traitement des erreurs lexicales

Les erreurs lexicales étant plus rares que les erreurs syntaxiques, la correction des erreurs est moins élaborée. Elle repose uniquement sur une correction locale, et utilise le même principe que dans l'analyse syntaxique. Mais seuls les modèles permettant l'insertion, le remplacement ou la suppression d'un caractère sont

autorisés. L'utilisateur a seulement la possibilité de modifier la longueur des modèles et l'ordre dans lequel ils sont définis dans la liste fournie par SYNTAX :

		Modèle	Interprétation
1	:	1 2 3 4	suppression
2	:	X 1 2 3 4	remplacement
3	:	X 0 1 2 3	insertion

Au niveau lexical, il est impossible d'agir sur le caractère précédent le caractère en erreur, 0 désigne donc le caractère en erreur, 1 le caractère qui le suit etc... Si aucun modèle ne s'applique, le caractère en erreur est supprimé.

6.3 Le fichier standard.recor

Ce fichier permet non seulement de spécifier le traitement des erreurs des niveaux lexicographique et syntaxique et également les textes des diagnostics qui seront produits. Il est donc possible, par exemple, de décrire les messages en Français, Allemand ou patois...

La spécification contenue dans `standard.recor` est donnée en annexe D.

Le texte d'un message est formé d'une séquence de chaînes de caractères (les guillemets internes doivent être précédés de "\"). "$\$n$, où n est un entier positif ou nul, sera remplacé par le n-ième symbole de la chaîne source (attention, dans l'analyseur lexical, le caractère en erreur est le numéro 0, alors que dans l'analyseur syntaxique le terminal en erreur est le numéro 1). "%n" sera remplacé par le symbole correspondant au "X" (ou au "S") d'indice n dans le modèle courant. Ainsi avec le modèle `0 X 1 X 3`, "%1" désigne le premier X et "%3" le second.

"`Dont_Delete`" et "`Dont_Insert`" sont des ensembles de symboles, spécifiés par l'utilisateur, qui permettent d'influer sur le mécanisme général qui vient d'être décrit, afin de limiter les possibilités de suppression ou d'insertion de symboles à forte connotation "sémantique". Il est en effet souvent dangereux pour la suite de l'analyse de supprimer ou d'insérer un "symbole ouvrant" qui nécessite que l'on rencontre plus tard le "symbole fermant" correspondant. La prise en compte de ces deux ensembles conduit à une modification dynamique des priorités respectives des différentes instances des modèles de correction. Ils ont en effet la définition suivante :

> Lors d'une tentative de correction, une instance d'un modèle qui implique la suppression d'un symbole de `Dont_Delete` ne sera choisie que s'il n'existe pas d'instance d'un modèle, même moins prioritaire, qui ne nécessite ni la suppression d'un élément de `Dont_Delete` ni l'insertion d'un élément de `Dont_Insert`. De même, l'insertion d'un élément de `Dont_Insert` ne pourra avoir lieu que si aucun modèle ne permet de correction sans suppression d'élément de `Dont_Delete`.

Pour le niveau lexical, les éléments de ces ensembles sont des caractères ; pour le niveau syntaxique, il s'agit de symboles terminaux.

"`Key_Terminals`", dans la spécification destinée à la récupération globale de l'analyseur syntaxique, définit l'ensemble des symboles terminaux sur lesquels sera tentée la récupération si la correction a échoué. Ces symboles sont également utilisés pour limiter la taille d'un modèle de correction. Soit " `0 X 1 2 3 4` "

un modèle tel que, pour l'erreur courante, le symbole a_2 soit un élément de "`Key_Terminals`". Dans ce cas, il est équivalent au modèle " `0 X 1 2` ", la chaîne a_3a_4 n'a pas besoin d'être validée. Cette extension permet d'accroître les possibilités de correction dans des portions de texte denses en erreurs.

Chapitre 7

TABLES_C : Production des tables d'analyse en langage C

Le module TABLES_C constitue l'épilogue obligatoire d'une construction ; il a deux fonctions :

- Vérifier la cohérence globale des tables qui ont été préalablement produites par les différents constructeurs de SYNTAX—ces tables contiennent en effet une indication permettant de déterminer si les constructeurs ont exploité la même version de la grammaire syntaxique.
- Traduire l'ensemble de ces tables en un programme en langage C formé essentiellement de variables initialisées.

TABLES_C produit un texte sur sa sortie standard (`stdout`), qui est usuellement redirigée vers un fichier de nom *nom-du-langage*`_t.c` ; ce texte rassemble, pour le langage *nom-du-langage*, toutes les tables nécessaires à l'analyse et au traitement de la sémantique, produites par les diverses phases de la construction. Il contient en outre, si *nom-du-langage* comporte des mots clés, le texte, en langage C, de la fonction `check_keyword`.

Chapitre 8

Analyse d'un texte source

L'analyse d'un texte source est réalisée en une ou deux passes, suivant le type de description sémantique choisi.

La première passe correspond à l'analyse lexicale et syntaxique du texte source.

L'analyse lexicale est réalisée par l'analyseur lexical—ou *scanner*, qui est un interpréteur de l'automate d'états finis produit par LECL. Le *scanner* lit le texte source (par l'intermédiaire du *source manager*) et remplit d'une part la structure `terminal_token` qui sera exploitée par l'analyseur syntaxique et d'autre part la `string_table` (table contenant les textes des unités lexicales, exploitée par l'intermédiaire du *string manager*).

L'analyse syntaxique est réalisée par l'analyseur syntaxique—ou *parser*, qui est un interpréteur de l'automate à pile produit par CSYNT. Au cours de cette passe, selon le type de sémantique choisi, des appels à des actions sémantiques sont effectués ou bien un arbre abstrait est engendré.

Dans le cas où on réalise la sémantique par arbre abstrait, une deuxième passe (*SEMANTIC PASS*) est effectuée. Elle consiste uniquement à exécuter le programme `semantic_pass` écrit par l'utilisateur.

Les analyseurs lexical et syntaxique contiennent également des modules permettant la correction *vel* la récupération automatique des erreurs. Ces modules ont été spécifiés par l'intermédiaire de RECOR—voir le chapitre 6.

8.1 Mise en œuvre

Se reporter à la documentation en ligne ; voir notamment `sxsrc_mngr(3)`, `sxscanner(3)`, `sxparser(3)`, `sxstr_mngr(3)`, `sxatc(3)`, `sxat_mngr(3)`.

Annexe A

Spécifications du langage BNF

A.1 Définition syntaxique

Ceci est la grammaire de BNF (voir le chapitre 2). Les chaînes de caractères suivant une règle de grammaire—à la droite du “ ;”—sont une spécification d'arbre abstrait destinée à SEMAT (voir en 5.2).

```
*
* This grammar is used for the specification of BNF
*
<BNF>              = <RULE_LIST>                        ; "BNF"
*
<RULE_LIST>        = <RULE_LIST>
                     <RULE>                             ;
<RULE_LIST>        = <RULE>                             ; "RULE_S"
*
<RULE>             = <VOCABULARY_LIST>  ";"             ;
*
<VOCABULARY_LIST> = <VOCABULARY_LIST>  <VOCABULARY>    ;
<VOCABULARY_LIST> = %LHS_NON_TERMINAL  =               ; "VOCABULARY_S"
*
<VOCABULARY>       = %NON_TERMINAL                      ; "NON_TERMINAL"
<VOCABULARY>       = %NON_TERMINAL  %PREDICATE          ; "X_NON_TERMINAL"
<VOCABULARY>       = %ACTION                            ; "ACTION"
<VOCABULARY>       = <TERMINAL>                         ;
<VOCABULARY>       = <TERMINAL>  %PREDICATE             ; "X_TERMINAL"
*
<TERMINAL>         = %TERMINAL                          ; "TERMINAL"
<TERMINAL>         = %GENERIC_TERMINAL                  ; "GENERIC_TERMINAL"
```

A.2 Définition lexicale

```
Classes

        PRINT           = ANY - #000..#040 - #177 ;
```

```
        T_HEADER         = PRINT - "<#;\"~&@" ;
        NT_BODY          = PRINT + SP - ">" ;
        OCTAL            = "0".."7" ;
```

Abbreviations

```
        NUMBER_NORMAL_FORM
                         = @Mark {"0"&True}+ @Erase {DIGIT}+
                         | "0"&True
                         | {DIGIT}+ ;
```

Tokens

```
        Comments         = -{SP | HT | VT | FF |
                             "~" {^"~\n" | {EOL}+ ^"<~\n"} {EOL} "~" |
                             EOL {"*" {^EOL} EOL}}+ ;
                            Priority Shift>Reduce ;
                            Context All But Comments ;
        %LHS_NON_TERMINAL
                         = "<"&Is_First_Col @Set (0) {NT_BODY} ">" ;
        %NON_TERMINAL    = "<" {NT_BODY} ">" ;
        %TERMINAL        = (-"#" PRINT | T_HEADER) {PRINT}
                         | -QUOTE {^"\"\n\\" |
                                   -"\\" {-EOL -"\\"}
                                       (-EOL ^"\"\n\\<" |
                                        -"n" @Put (EOL) |
                                        -"b" @Put (BS) |
                                        -"t" @Put (HT) |
                                        -"v" @Put (VT) |
                                        -"f" @Put (FF) |
                                        -"r" @Put (CR) |
                                        @Mark OCTAL&True
                                             [OCTAL&True [OCTAL&True]]
                                             @2 |
                                        ^"bnfrtv\n"
                                       )
                                  }+ {-"\\" -EOL}
                           -QUOTE ;
                            Priority Shift>Reduce ;
                            Context Comments ;
        %GENERIC_TERMINAL
                         = "%" LETTER {["_"] (LETTER | DIGIT)} ;
                            Priority Reduce>Reduce, Shift>Reduce ;
                            Context Comments ;
        %ACTION          = "@" NUMBER_NORMAL_FORM ;
                            Context Comments ;
        %PREDICATE       = "&" NUMBER_NORMAL_FORM ;
                            Context Comments ;
        "="              = -"="&Is_Set (0) @Reset (0) ;
        ";"              = -";" @1 ;
```

```
--  scan_act (INIT)  : erases the source text from beginning until the
--                         first "<" laying at column 1
--  @1  : Skip the source text until the first "<" laying at column 1
--  @2  : Convert octal code \nnn to character
```

A.3 Actions et prédicats lexicaux

Ceci est le codage en langage C des actions et prédicats associés à la description lexicale de BNF.

```
#include "sxunix.h"


#define rule_slice 100

static struct span {
           unsigned long        head;
           struct tail {
               unsigned long    line;
               unsigned int     column;
           }    tail;
       }        *coords;
static int      xprod, rules_no;
static int      current_rule_no;



int     bnf_get_line_no (rule_no)
    int         rule_no;
{
    return rule_no > xprod ? 0 : coords [rule_no].head;
}



SXVOID  bnf_skip_rule ()
{
    register struct tail        *tail_coord;
    register unsigned long      line;
    register unsigned int       column;

    if (current_rule_no >= xprod) {
        current_rule_no = 0;
    }

    tail_coord = &coords [++current_rule_no].tail;
    line = tail_coord->line;
    column = tail_coord->column;

    while (sxsrcmngr.source_coord.line < line) {
        if (sxnext_char () == EOF)
            return;
    }
```

```
    while (sxsrcmngr.source_coord.column < column) {
        if (sxnext_char () == EOF)
            return;
    }
}



SXVOID  bnf_found_bad_beginning_of_rule ()
{
    struct sxsource_coord       less_coord;

    less_coord = sxsrcmngr.source_coord, less_coord.column = 1;
    sxput_error
         (less_coord,
          "%s\
Illegal occurrence of \"<\"; \
the head of a new grammar rule is assumed.",
          sxsvar.sxtables->err_titles [2]);
    sxsrcpush ('\n', "<", less_coord);
}



static VOID     gripe ()
{
    fputs ("\nThe function \"bnf_scan_act\" is out of date \
with respect to its specification.\n", sxstderr);
    abort ();
}



SXVOID  (*more_scan_act) ();



bnf_scan_act (code, act_no)
    int         code;
    int         act_no;
{
    switch (code) {
    case OPEN:
        coords = (struct span*) sxalloc (rules_no = rule_slice,
                                         sizeof (struct span))
                                - 1;

        if (more_scan_act != NULL) {
            (*more_scan_act) (code, act_no);
        }

        return;
```

```
    case CLOSE:
        if (more_scan_act != NULL) {
            (*more_scan_act) (code, act_no);
        }

        if (coords != NULL) {
            sxfree (coords + 1);
        }

        return;

    case INIT:
        if (more_scan_act != NULL) {
            (*more_scan_act) (code, act_no);
        }

        {
            register SHORT      c;

            c = sxsrcmngr.current_char;

            while (c != EOF)
                if (c == '<' && sxsrcmngr.source_coord.column == 1) {
                    coords [xprod = 1].head =
                         sxsrcmngr.source_coord.line;
                    return;
                }
                else
                    c = sxnext_char ();
        }

        fprintf (sxstderr, "\n\tThis text is not a grammar... \
See you later.\n");
        SXEXIT (3);

    case FINAL:
        if (more_scan_act != NULL) {
            (*more_scan_act) (code, act_no);
        }

        return;

    case ACTION:
        switch (act_no) {
            register SHORT      c;

        case 1:
            coords [xprod].tail.line = sxsrcmngr.source_coord.line;
            coords [xprod].tail.column = sxsrcmngr.source_coord.column;

            if (more_scan_act != NULL) {
                (*more_scan_act) (code, act_no);
            }
```

```
            c = sxsrcmngr.current_char;

            while (c != EOF)
                if (c == '<' && sxsrcmngr.source_coord.column == 1) {
                    if (++xprod > rules_no) {
                        coords = (struct span*)
                              sxrealloc (coords + 1,
                                         rules_no += rule_slice,
                                         sizeof (struct span))
                              - 1;
                    }

                    coords [xprod].head = sxsrcmngr.source_coord.line;
                    return;
                }
                else
                    c = sxnext_char ();

            return;

        case 2:
            /* \nnn => char */
            {
                register int    val;
                register char   c, *s, *t;

                t = s = sxsvar.lv_s.token_string + sxsvar.lv.mark.index;

                for (val = *s++ - '0'; (c = *s++) != NUL; ) {
                    val = (val << 3) + c - '0';
                }

                *t = val;
                sxsvar.lv.ts_lgth = sxsvar.lv.mark.index + 1;
                sxsvar.lv.mark.index = -1;
            }

            return;
        }

    default:
        gripe ();
    }
}
```

Annexe B

Spécifications du langage LECL

B.1 Définition syntaxique

Ceci est la grammaire de LECL (voir le chapitre 4). Les chaînes de caractères suivant une règle de grammaire—à la droite du ";"—sont une spécification d'arbre abstrait destinée à SEMAT (voir en 5.2). Les commandes entre tildes ("~") spécifient un paragrapheur du langage LECL et doivent être ignorées du point de vue strictement grammatical (cf `paradis(1)` et le rapport INRIA N° 455, intitulé *Paradis, un Système de Paragraphage Dirigé par la Syntaxe*, Novembre 1985).

```
<LECL_GRAMMAR>  = <CLASSES_LIST_OPTION>


                  <ABBREVIATIONS_LIST_OPTION>


                  <TOKENS_LIST_OPTION>


                  <SYNONYMS_LIST_OPTION>


                  <REPR_SPEC_LIST_OPTION>
                                              ~~; "PROGRAM_ROOT_LECL"
*
*    C L A S S E S
*
<CLASSES_LIST_OPTION>
                = ~INH~                       ~~;
<CLASSES_LIST_OPTION>
                = CLASSES

                ~TAB~   <CLASSES_LIST>  ";"     ~~;
```

```
<CLASSES_LIST>  = <CLASSES_LIST>  ";"
                  <CLASS>                       ~~;
<CLASSES_LIST>  = <CLASS>                       ~~; "CLASS_S"

<CLASS>         = <CLASS_NAME> ~
                  COL(24)~ ~SPACE~ "="  <CLASS_EXP> ~
                                                 ~; "CLASS"

<CLASS_NAME>    = %IDENTIFIER                   ~~; "CLASS_NAME"

<CLASS_EXP>     = <CLASS_EXP>  +  <CLASS_TERM>  ~~; "PLUS"
<CLASS_EXP>     = <CLASS_EXP>  -  <CLASS_TERM>  ~~; "MINUS"
<CLASS_EXP>     = <CLASS_TERM>                  ~~;

<CLASS_TERM>    = ( <CLASS_EXP> )               ~~;
<CLASS_TERM>    = <CLASS_REF>                   ~~;

<CLASS_REF>     = <SIMPLE_REF>                  ~~;
<CLASS_REF>     = <SLICE>                       ~~;

<SLICE>         = <SIMPLE_REF> .. <SIMPLE_REF>  ~~; "SLICE"

<SIMPLE_REF>    = %IDENTIFIER                   ~~; "ID"
<SIMPLE_REF>    = %STRING_LITERAL               ~~; "STRING"
<SIMPLE_REF>    = %OCTAL_CODE                   ~~; "OCTAL_CODE"
*
*   A B B R E V I A T I O N S
*
<ABBREVIATIONS_LIST_OPTION>
                = ~INH~                         ~~;
<ABBREVIATIONS_LIST_OPTION>
                = ABBREVIATIONS

                ~TAB~ <ABBREVIATIONS_LIST>  ";" ~~;

<ABBREVIATIONS_LIST>
                = <ABBREVIATIONS_LIST>  ";"
                  <ABBREVIATION>                ~~;
<ABBREVIATIONS_LIST>
                = <ABBREVIATION>                ~~; "ABBREVIATION_S"

<ABBREVIATION>  = <REGULAR_EXPRESSION_NAME> ~
                  COL(24)~ ~SPACE~ "="  <REGULAR_EXPRESSION> ~
                                                 ~; "ABBREVIATION"
<ABBREVIATION>  = <PREDICATE_NAME> ~
                  COL(24)~ ~SPACE~ "="  <&_EXPRESSION> ~
                                                 ~; "ABBREVIATION"

<REGULAR_EXPRESSION_NAME>
                = %IDENTIFIER                   ~~;
                                                  "ABBREVIATION_RE_NAME"

<PREDICATE_NAME>= %PREDICATE_NAME               ~~;
                                               "ABBREVIATION_PRDCT_NAME"
```

```
<&_EXPRESSION>  = <&_OR>                        ~~; "PRDCT_EXPRESSION"

<&_OR>          = <&_OR>  |  <&_ET>             ~~; "PRDCT_OR"
<&_OR>          = <&_ET>                        ~~;

<&_ET>          = <&_ET>  <&_TERM>              ~~; "PRDCT_ET"
<&_ET>          = <&_TERM>                      ~~;

<&_TERM>        = ( <&_OR> )                    ~~; "EXPRESSION"
<&_TERM>        = %PREDICATE_NAME               ~~;
<&_TERM>        = <DYNAMIC_PREDICATE>           ~~;
<&_TERM>        = ^ <DYNAMIC_PREDICATE>         ~~; "PRDCT_NOT"
*
*   T O K E N S
*
<TOKENS_LIST_OPTION>
                = ~INH~                         ~~;
<TOKENS_LIST_OPTION>
                = TOKENS

                ~TAB~   <TOKENS_LIST>           ~~;

<TOKENS_LIST>   = <TOKENS_LIST>
                  <TOKEN>                       ~~;
<TOKENS_LIST>   = <TOKEN>                       ~~; "TOKEN_S"

<TOKEN>         = <TOKEN_DEF>  ";" <VOID>       ~~; "TOKEN"
<TOKEN>         = <TOKEN_DEF>  ";"
                  ~COL(27)~ <ENVIRONMENT_LIST>  ~~; "TOKEN"

<TOKEN_DEF>     = <LEXICAL_UNIT_NAME> ~
                  COL(24)~ ~SPACE~ "="  <TOKEN_BODY> ~
                                                 ~; "TOKEN_DEF"

<LEXICAL_UNIT_NAME>
                = %GENERIC_NAME                 ~~; "LEXICAL_UNIT_NAME"
<LEXICAL_UNIT_NAME>
                = %STRING_LITERAL               ~~; "LEXICAL_UNIT_NAME"
<LEXICAL_UNIT_NAME>
                = %IDENTIFIER                   ~~; "LEXICAL_UNIT_NAME"
<LEXICAL_UNIT_NAME>
                = EOF                           ~~; "EOF"
<LEXICAL_UNIT_NAME>
                = COMMENTS                      ~~; "COMMENTS"
<LEXICAL_UNIT_NAME>
                = INCLUDE                       ~~; "INCLUDE"

<TOKEN_BODY>    = <REGULAR_EXPRESSION>          ~~;
<TOKEN_BODY>    =
                  ~COL(14)~ <UNION>             ~~;

<UNION>         = UNION
                     <COMPONENT_LIST>
```

```
                 END                           ~~;

<COMPONENT_LIST>= <COMPONENT_LIST>
                  <COMPONENT>                  ~~;
<COMPONENT_LIST>= <COMPONENT>                  ~~; "COMPONENTS_S"

<COMPONENT>     = <COMPONENT_DEF>  ";" <VOID>   ~~; "COMPONENT"
<COMPONENT>     = <COMPONENT_DEF>  ";"
                  ~COL(27)~ <ENVIRONMENT_LIST>  ~~; "COMPONENT"

<COMPONENT_DEF> = <COMPONENT_NAME> ~
                  COL(24)~ ~SPACE~ :  <REGULAR_EXPRESSION> ~
                                               ~; "COMPONENT_DEF"

<COMPONENT_NAME>= %IDENTIFIER                  ~~; "COMPONENT_NAME_DEF"

<ENVIRONMENT_LIST>
                = <ENVIRONMENT_LIST>
                  <ENVIRONMENT>  ";"           ~~;
<ENVIRONMENT_LIST>
                = <ENVIRONMENT>  ";"           ~~; "ENVIRONMENT_S"

<ENVIRONMENT>=  %IDENTIFIER &1  %IDENTIFIER ~ ~ ; "NOT_KEYWORD"
* &1 retourne VRAI ssi le premier identificateur est NOT
*                      et le second KEYWORD
<ENVIRONMENT>   =  %ACTION_NO                  ~~; "POST_ACTION"
<ENVIRONMENT>   =  <PRIORITY>                  ~~;
<ENVIRONMENT>   =  <CONTEXT>                   ~~;

<PRIORITY>      = PRIORITY  <PRIORITY_KIND_LIST>~~;

<PRIORITY_KIND_LIST>
                = <PRIORITY_KIND_LIST> ,  <PRIORITY_KIND> ~
                                               ~;
<PRIORITY_KIND_LIST>
                = <PRIORITY_KIND>              ~~; "PRIORITY_KIND_S"

<PRIORITY_KIND> = REDUCE > REDUCE              ~~; "REDUCE_REDUCE"
<PRIORITY_KIND> = REDUCE > SHIFT               ~~; "REDUCE_SHIFT"
<PRIORITY_KIND> = SHIFT > REDUCE               ~~; "SHIFT_REDUCE"

<VOID>          =                              ~~;

<CONTEXT>       = CONTEXT  <TOKEN_REF_LIST>     ~~; "CONTEXT"
<CONTEXT>       = CONTEXT  ALL  BUT  <TOKEN_REF_LIST> ~
                                               ~; "RESTRICTED_CONTEXT"
<CONTEXT>       = CONTEXT  ALL <VOID>           ~~; "RESTRICTED_CONTEXT"
<CONTEXT>       = UNBOUNDED  CONTEXT  <TOKEN_REF_LIST> ~
                                               ~; "UNBOUNDED_CONTEXT"
<CONTEXT>       = UNBOUNDED  CONTEXT  ALL  BUT  <TOKEN_REF_LIST> ~
                                               ~;
                                         "UNBOUNDED_RESTRICTED_CONTEXT"
<CONTEXT>       = UNBOUNDED  CONTEXT  ALL <VOID>~~;
                                         "UNBOUNDED_RESTRICTED_CONTEXT"
```

```
<TOKEN_REF_LIST>= <TOKEN_REF_LIST> ,  <TOKEN_REF> ~
                                                ~;
<TOKEN_REF_LIST>= <TOKEN_REF>                   ~~; "TOKEN_REF_S"

<TOKEN_REF>     = <TOKEN_NAME> <VOID>           ~~; "COMPONENT_REF"
<TOKEN_REF>     = <TOKEN_NAME> . <COMPONENT_NAME_REF> ~
                                                ~; "COMPONENT_REF"

<TOKEN_NAME>    = COMMENTS                      ~~; "CONTEXT_COMMENTS"
<TOKEN_NAME>    = EOF                           ~~; "CONTEXT_EOF"
<TOKEN_NAME>    = INCLUDE                       ~~; "CONTEXT_NAME"
<TOKEN_NAME>    = %IDENTIFIER                   ~~; "CONTEXT_NAME"
<TOKEN_NAME>    = %GENERIC_NAME                 ~~; "CONTEXT_NAME"
<TOKEN_NAME>    = %STRING_LITERAL               ~~; "CONTEXT_NAME"

<COMPONENT_NAME_REF>
                = %IDENTIFIER                   ~~; "COMPONENT_NAME_REF"
*
*   R E G U L A R   E X P R E S S I O N S
*
<REGULAR_EXPRESSION>
                = <ALTERNATIVE>                 ~~; "REGULAR_EXPRESSION"

<ALTERNATIVE>   = <ALTERNATIVE>  |  <SEQUENCE>  ~~; "ALTERNATIVE"
<ALTERNATIVE>   = <SEQUENCE>                    ~~;

<SEQUENCE>      = <SEQUENCE>  <TERM>            ~~; "SEQUENCE"
<SEQUENCE>      = <TERM>                        ~~;

<TERM>          = - <ITEM>                      ~~; "ERASE"
<TERM>          = <ITEM>                        ~~;

<ITEM>          = ( <ALTERNATIVE> )             ~~; "EXPRESSION"
<ITEM>          = [ <ALTERNATIVE> ]             ~~; "OPTION"
<ITEM>          = { <ALTERNATIVE> }             ~~; "REF_TRANS_CLOSURE"
<ITEM>          = { <ALTERNATIVE> } *           ~~; "REF_TRANS_CLOSURE"
<ITEM>          = ( <ALTERNATIVE> ) *           ~~; "REF_TRANS_CLOSURE"
<ITEM>          = { <ALTERNATIVE> } +           ~~; "TRANS_CLOSURE"
<ITEM>          = ( <ALTERNATIVE> ) +           ~~; "TRANS_CLOSURE"
<ITEM>          = <EXTENDED_CLASS_REF>          ~~;
<ITEM>          = <ACTION>                      ~~;

<EXTENDED_CLASS_REF>
                = <NOT_CLASS_REF> <PREDICATE>   ~~; "EXTENDED_CLASS_REF"
<EXTENDED_CLASS_REF>
                = <NOT_CLASS_REF>               ~~;

<NOT_CLASS_REF> = ^ <CLASS_REF>                 ~~; "NOT"
<NOT_CLASS_REF> = <CLASS_REF>                   ~~;

<ACTION>        = %ACTION_NO                    ~~;
<ACTION>        = "@UPPER_CASE"                 ~~; "UPPER_CASE"
<ACTION>        = "@LOWER_CASE"                 ~~; "LOWER_CASE"
```

```
<ACTION>        = "@LOWER_TO_UPPER"              ~~; "LOWER_TO_UPPER"
<ACTION>        = "@UPPER_TO_LOWER"              ~~; "UPPER_TO_LOWER"
<ACTION>        = "@ERASE"                       ~~; "ACTION_ERASE"
<ACTION>        = "@SET"  ( %INTEGER_NUMBER )    ~~; "SET"
<ACTION>        = "@RESET"  ( %INTEGER_NUMBER )  ~~; "RESET"
<ACTION>        = "@INCR"  ( %INTEGER_NUMBER )   ~~; "INCR"
<ACTION>        = "@DECR"  ( %INTEGER_NUMBER )   ~~; "DECR"
<ACTION>        = "@PUT"  ( <SIMPLE_REF> )       ~~; "PUT"
<ACTION>        = "@MARK"                        ~~; "MARK"
<ACTION>        = "@RELEASE"                     ~~; "RELEASE"

<PREDICATE>     = %PREDICATE_NAME                ~~;
<PREDICATE>     = <STATIC_PREDICATE>             ~~;
<PREDICATE>     = <DYNAMIC_PREDICATE>            ~~;

<STATIC_PREDICATE>
                = "&TRUE"                        ~~; "IS_TRUE"
<STATIC_PREDICATE>
                = "&FALSE"                       ~~; "IS_FALSE"

<DYNAMIC_PREDICATE>
                = %PREDICATE_NO                  ~~;
<DYNAMIC_PREDICATE>
                = "&IS_FIRST_COL"                ~~; "IS_FIRST_COL"
<DYNAMIC_PREDICATE>
                = "&IS_LAST_COL"                 ~~; "IS_LAST_COL"
<DYNAMIC_PREDICATE>
                = "&IS_SET"  ( %INTEGER_NUMBER )~~; "IS_SET"
<DYNAMIC_PREDICATE>
                = "&IS_RESET"  ( %INTEGER_NUMBER ) ~
                                                  ~; "IS_RESET"
*
*  S Y N O N Y M S
*
<SYNONYMS_LIST_OPTION>
                = ~INH~                          ~~;
<SYNONYMS_LIST_OPTION>
                = SYNONYMS

                  ~TAB~ <SYNONYMS_LIST>  ";"     ~~;

<SYNONYMS_LIST> = <SYNONYMS_LIST>  ";"
                  <DENOTATION_LIST>              ~~;
<SYNONYMS_LIST> = <DENOTATION_LIST>              ~~; "SYNONYM_S"

<DENOTATION_LIST>
                = <DENOTATION_LIST> ,  <DENOTATION> ~
                                                  ~;
<DENOTATION_LIST>
                = <DENOTATION> ~
                  COL(24)~ ~SPACE~ "="  <DENOTATION> ~
                                                  ~; "DENOTATION_S"

<DENOTATION>    = %IDENTIFIER                    ~~; "ID_DENOTATION"
```

```
<DENOTATION>    = %STRING_LITERAL               ~~; "STRING_DENOTATION"
*
* R E P R E S E N T A T I O N   S P E C I F I C A T I O N
*
<REPR_SPEC_LIST_OPTION>
                = ~INH~                         ~~;
<REPR_SPEC_LIST_OPTION>
                = ~TAB~ <REPR_SPEC_LIST>  ";"   ~~;

<REPR_SPEC_LIST>= <REPR_SPEC_LIST>  ";"
                  <REPR_SPEC>                   ~~;
<REPR_SPEC_LIST>= <REPR_SPEC>                   ~~; "REPR_SPEC_S"

<REPR_SPEC>     = <COLLATING_LIST_REPR>         ~~;
<REPR_SPEC>     = <BYTE_LENGTH_REPR>            ~~;
<REPR_SPEC>     = <WORD_LENGTH_REPR>            ~~;

<COLLATING_LIST_REPR>
                = FOR  INTERNAL  CODE  USE
                        <COLLATING_LIST>        ~~;

<BYTE_LENGTH_REPR>
                = FOR  BYTE  USE  %INTEGER_NUMBER  BITS ~
                                                 ~; "BYTE_LENGTH"

<WORD_LENGTH_REPR>
                = FOR  WORD  USE  %INTEGER_NUMBER  BITS ~
                                                 ~; "WORD_LENGTH"

<COLLATING_LIST>= <COLLATING_LIST>
                ,  <COMPOSANT>                  ~~;
<COLLATING_LIST>= <COMPOSANT>                   ~~; "COLLATING_S"

<COMPOSANT>     = <CLASS_REF>                   ~~;
<COMPOSANT>     = UNUSED                        ~~; "NOT_SPECIFIED"
<COMPOSANT>     = %INTEGER_NUMBER  UNUSED       ~~;
```

B.2 Définition lexicale

```
Classes
        OCTAL           = "0".."7" ;


Abbreviations

        IDENT           = LETTER {["_"] (LETTER | DIGIT)} ;
        DARK_LET        = LETTER -{BS @1 LETTER}+ ;
        BOLD_LET        = LETTER -{BS @1 LETTER} ;
        BOLD_US         = "_" -{BS "_"} ;
        DARK_US         = "_" -{BS "_"}+ ;
        BOLD_COMMERCIAL_AT
                        = "@" -{BS "@"} ;
        DARK_AMPERSAND  = "&" -{BS "&"}+ ;
```

```
BOLD_WORD        = BOLD_LET {[BOLD_US] BOLD_LET} ;
DARK_WORD        = DARK_LET {[DARK_US] DARK_LET} ;
NUMBER_NORMAL_FORM
                 = @Mark {"0"&True}+ @Erase {DIGIT}+
                 | "0"&True
                 | {DIGIT}+ ;
```

Tokens

```
Comments         = -({SP | HT | EOL | VT | FF}+
                 | "-" "-" {^EOL} EOL) ;
                    Context All But Comments ;
%IDENTIFIER      = IDENT @Upper_Case ;
                    Context All But %IDENTIFIER, KEYWORD,
                         %INTEGER_NUMBER ;
%PREDICATE_NAME = "&" IDENT @Upper_Case ;
                    Context All But %IDENTIFIER ;
%STRING_LITERAL = QUOTE
                   {
                    ^"\"\n" |
                    -"\\"&True
                         (-EOL&True |
                          -"n"&True @Put (EOL) |
                          -"b"&True @Put (BS) |
                          -"t"&True @Put (HT) |
                          -"v"&True @Put (VT) |
                          -"f"&True @Put (FF) |
                          -"r"&True @Put (CR) |
                          @Mark OCTAL&True
                               [OCTAL&True [OCTAL&True]]
                               @2 |
                          ANY)
                   }+
                  QUOTE ;
                    Context All But %STRING_LITERAL ;
%OCTAL_CODE      = "#" {OCTAL}+ ;
%GENERIC_NAME    = "%" IDENT @Upper_Case ;
%INTEGER_NUMBER = {DIGIT}+ ;
%PREDICATE_NO    = "&" NUMBER_NORMAL_FORM ;
%ACTION_NO       = "@" NUMBER_NORMAL_FORM ;
KEYWORD          = {DARK_LET}+ @Upper_Case ;
                    Context All But %IDENTIFIER, KEYWORD ;
ACTION_KEYWORD   = BOLD_COMMERCIAL_AT BOLD_WORD @Upper_Case ;
                    Context All But %IDENTIFIER ;
PREDICATE_KEYWORD
                 = DARK_AMPERSAND DARK_WORD @Upper_Case ;
                    Context All But %IDENTIFIER ;
"-"              = -"-" ;
                    Priority Shift>Reduce ;
```

Synonyms

```
        "EOF"           = "END_OF_FILE" ;

-- @1 : Is_The_Same_Letter
-- @2 : \nnn => char
```

B.3 Actions lexicales

Ceci est le codage en langage C des actions (il n'y a pas de prédicat) associées à la description lexicale de LECL.

```
#include "sxunix.h"

VOID    lecl_scan_act (entry, act_no)
    int         entry, act_no;
{
    switch (entry) {
    case OPEN:
    case CLOSE:
    case INIT:
    case FINAL:
        return;

    case ACTION:
        switch (act_no) {
        case 1:
            /* Dark Letter Check */
            if (sxsrcmngr.current_char
                 !=
                 sxsvar.lv_s.token_string [sxsvar.lv.ts_lgth - 1])
                sxput_error
                     (sxsrcmngr.source_coord,
                      "%s\
A dark symbol must be built up with the same character.",
                      sxsvar.sxtables->err_titles [1] /* Warning */ );

            return;

        case 2:
            /* \nnn => char */
            {
                register int    val;
                register char   c, *s, *t;

                t = s = sxsvar.lv_s.token_string + sxsvar.lv.mark.index;

                for (val = *s++ - '0'; (c = *s++) != NUL; ) {
                    val = (val << 3) + c - '0';
                }

                *t = val;
                sxsvar.lv.ts_lgth = sxsvar.lv.mark.index + 1;
                sxsvar.lv.mark.index = -1;
            }
```

```
            return;

        default:
            break;
        }

    default:
        fputs ("The function \"lecl_scan_act\" is out of date \
with respect to its specification.\n", sxstderr);
        abort ();
    }
}
```

B.4 Prédicats syntaxiques

Ceci est le codage en langage C des prédicats syntaxiques (il n'y a pas d'action) associés à la description syntaxique de LECL.

```
#include "sxunix.h"

static int NOT_code, KEYWORD_code;

int lecl_pars_act (entry, action_no)
    int entry, action_no;
{
    switch (entry) {
    case OPEN:
    case CLOSE:
    case FINAL:
return;

    case INIT:
/* The keywords "NOT" and "KEYWORD" are not reserved. */
NOT_code = sxstrsave ("NOT");
KEYWORD_code = sxstrsave ("KEYWORD");
return;

    case PREDICATE:
switch (action_no) {
case 1:
    return sxget_token (sxplocals.ptok_no)->string_table_entry
            == NOT_code && sxget_token (sxplocals.ptok_no +
            1)->string_table_entry == KEYWORD_code;

default:
    break;
}

break;

    default:
break;
```

```
    }

    fputs ("The function \"lecl_pars_act\" is out of date \
with respect to its specification.\n", sxstderr);
    abort ();
}
```

Annexe C

Exemples liés à la sémantique

C.1 Le programme d'actions sémantiques

On trouvera ci-dessous la structure que doit respecter le programme utilisateur codant les actions sémantiques.

```
#include "sxunix.h"

static  action (action_no)
    int         action_no;
{
    switch (action_no) {
    case 0:
        break;

    case 1:
        ...
        break;

    case 2:
        ...
        break;
    .
    .
    .
    default:
        /* Message d'erreur */
    }
}


L_action (entry, arg)
    int         entry;
    struct sxtables     *arg;
{
    switch (entry) {
    case OPEN:  /* appele avant l'analyse des textes source. On prepare
                   les donnees ne dependant que du langage source et
                   non des textes source */
```

```
        ...
        break;

    case INIT:  /* appele avant l'analyse d'un texte */
        ...
        break;

    case ACTION:/* appele a chaque reduction syntaxique */
        action (arg); /* arg == action_no !!!! */
        break;

    case ERROR: /* appele en cas d'erreur de syntaxe non corrigible */
        ...
        break;

    case FINAL: /* appele apres l'analyse syntaxique d'un texte */
        ...
        break;

    case SEMPASS: /* appele apres l'analyse syntaxique d'un texte*/
        ...
        break;

    case CLOSE: /* appele apres l'analyse de tous les textes source */
        ...
        break;

    default:
        /* Message d'erreur */
    }
}
```

C.2 Trame de passes sémantiques

Ceci est la trame de passe sémantique produite par SEMAT à partir de la spécification d'arbre abstrait de l'annexe A.1.

```
/* *****************************************************
   *  This program has been generated from bnf.at       *
   *  on Mon Apr 11 15:44:23 1988                       *
   *  by the SYNTAX (*) abstract tree constructor SEMAT  *
   *****************************************************
   *  (*) SYNTAX is a trademark of INRIA.               *
   ***************************************************** */



/*   I N C L U D E S   */
#define NODE struct bnf_node
#include "sxunix.h"

struct bnf_node {
    SXNODE_HEADER_S VOID_NAME;
```

```
/*
your attribute declarations...
*/
};

/*
N O D E   N A M E S
*/
#define ERROR_n 1
#define ACTION_n 2
#define BNF_n 3
#define GENERIC_TERMINAL_n 4
#define LHS_NON_TERMINAL_n 5
#define NON_TERMINAL_n 6
#define PREDICATE_n 7
#define RULE_S_n 8
#define TERMINAL_n 9
#define VOCABULARY_S_n 10
#define X_NON_TERMINAL_n 11
#define X_TERMINAL_n 12
/*
E N D   N O D E   N A M E S
*/


static bnf_pi () {

/*
I N H E R I T E D
*/

switch (VISITED->father->name) {
case ERROR_n :
break;

case BNF_n :/* VISITED->name = RULE_S_n */
break;

case RULE_S_n :/* VISITED->name = VOCABULARY_S_n */
break;

case VOCABULARY_S_n :/* VISITED->name = {ACTION_n, GENERIC_TERMINAL_n,
                        LHS_NON_TERMINAL_n, NON_TERMINAL_n,
                        TERMINAL_n, X_NON_TERMINAL_n, X_TERMINAL_n} */
break;

case X_NON_TERMINAL_n :
        switch (VISITED->position) {
        case 1 :/* VISITED->name = NON_TERMINAL_n */
        break;

        case 2 :/* VISITED->name = PREDICATE_n */
        break;
```

```
        }
break;

case X_TERMINAL_n :
        switch (VISITED->position) {
        case 1 :/* VISITED->name = {GENERIC_TERMINAL_n, TERMINAL_n} */
        break;

        case 2 :/* VISITED->name = PREDICATE_n */
        break;
        }
break;

/*
Z Z Z Z
*/
}
/* end bnf_pi */
}

static bnf_pd () {

/*
D E R I V E D
*/

switch (VISITED->name) {
case ERROR_n :
break;

case ACTION_n :
break;

case BNF_n :
break;

case GENERIC_TERMINAL_n :
break;

case LHS_NON_TERMINAL_n :
break;

case NON_TERMINAL_n :
break;

case PREDICATE_n :
break;

case RULE_S_n :
break;

case TERMINAL_n :
break;
```

```
case VOCABULARY_S_n :
break;

case X_NON_TERMINAL_n :
break;

case X_TERMINAL_n :
break;

/*
Z Z Z Z
*/
}
/* end bnf_pd */
}

static smpopen (sxtables_ptr)
struct sxtables *sxtables_ptr;
{
sxatcvar.atc_lv.node_size = sizeof (struct bnf_node);
}

static smppass ()
{

/*   I N I T I A L I S A T I O N S   */
/* ........... */

/*   A T T R I B U T E S   E V A L U A T I O N   */
sxsmp (sxatcvar.atc_lv.abstract_tree_root, bnf_pi, bnf_pd);

/*   F I N A L I S A T I O N S   */
/* ........... */

}

bnf_smp (what, sxtables_ptr)int what;
struct sxtables *sxtables_ptr;
{
switch (what) {
case OPEN:
smpopen (sxtables_ptr);
break;
case ACTION:
smppass ();
break;
}
}
```

Annexe D

Le fichier standard.recor

```
Titles
    "",
    "Warning:\t",
    "Error:\t";

Scanner
    Local
        1 2 3 4       ; "The invalid character \"" $0
                        "\" is deleted.";
        X 1 2 3 4     ; "The invalid character \"" $0
                        "\" is replaced by \"" %0 "\".";
        X 0 1 2 3     ; "The character \"" %0
                        "\" is inserted before \"" $0 "\".";

        Dont_Delete = {};
        Dont_Insert = {};

    Global
        Detection     : "\"%s\" is deleted.";
             -- parameter: character in error
        Keyword       : "This unknown keyword is erased.";
        Eol           : "End Of Line";
        Eof           : "End Of File";
        Halt          : "Scanning stops on End Of File.";

Parser
    Local
        0 S 2         ; "Misspelling of \"" $1
                        "\" which is replaced by the keyword \""
                        %1 "\".";
        S 1           ; "Misspelling of \"" $0 "\" before \"" $1
                        "\" which is replaced by the keyword \""
                        %0 "\".";
        0 X 1 2 3     ; "\"" %1 "\" is inserted before \"" $1 "\".";
        0 X 2 3 4     ; "\"" $1 "\" is replaced by \"" %1 "\".";
        0 2 3 4       ; "\"" $1 "\" is deleted.";
        0 X X 1 2 3 4 ; "\"" %1 " " %2 "\" is inserted before \"" $1
                        "\".";
```

```
        X 0 1 2 3     ; "\"" %0 "\" is inserted before \"" $0 " " $1
                        "\".";
        X 1 2 3 4     ; "\"" $0 "\" before \"" $1
                        "\" is replaced by \"" %0 "\".";
        1 2 3 4       ; "\"" $0 "\" before \"" $1 "\" is deleted.";
        X 2 3 4       ; "\"" $0 " " $1 "\" is replaced by \"" %0 "\".";
        X X 1 2 3     ; "\"" $0 "\" before \"" $1
                        "\" is replaced by \"" %0 " " %1 "\".";

        Dont_Delete = {};
        Dont_Insert = {};

    Forced_Insertion
        "\"" %0 "\" is forced before \"" $1 "\"." ;

    Global
        Key_Terminals = {};
        Validation_Length = 2;
        Followers_Number <= 5
                      : "\"%s\"^(, \"%s\"^) is expected";
             -- parameters: array (1:Followers_Number) of valid
             --             followers at detection point
        Detection     : "Global recovery.";
             -- parameters: none
        Restarting    : "Parsing resumes on \"%s\"";
             -- parameters: array (1:Validation_Length) of valid
             --             followers at restarting point
        Halt          : "Parsing stops on End Of File.";
             -- parameters: none

Abstract
    "%d errors and %d warnings are reported.";
         -- parameters: array (1:Titles_No) of number of messages
```

Annexe E

Exemples de grammaires non-LALR(1)

E.1 Une grammaire ambigüe

On trouvera ci-dessous une grammaire ambigüe illustrant le problème du “dangling else” et les messages de non conformité émis par CSYNT dans le cas standard (avec les options par défaut) et dans le cas ou l'option `-path` est positionnée.

E.1.1 Grammaire

```
<Stmt>         = <If_Stmt>                          ;
<Stmt>         =                                    ;

<If_Stmt>      = if cond <Then_Part> <Else_Part>  ;

<Then_Part>    = then <Stmt>                        ;

<Else_Part>    =                                    ;
<Else_Part>    = else <Stmt>                        ;
```

E.1.2 Listage des conflits

E.1.2.1 option par défaut

```
    N O T    L A L R (1)
Shift-Reduce conflict in state 6 on "else" between:
        - Shift:
            6: <Else_Part> = . "else" <Stmt>
        - Reduce:
            5: <Else_Part> = .
                derived from:
                    3: <If_Stmt> = "if" "cond" . <Then_Part> <Else_Part>

Using the system disambiguating rules, priority is given to shift.
```

E.1.2.2 Option -path

```
SAMPLE PATH from state 1 to 6

"if" "cond" <Then_Part>


    N O T   L R (1)
Shift-Reduce conflict in state 6 on "else" between:
        - Shift:
            6: <Else_Part> = . "else" <Stmt>
                derivation:
                    <Stmt>
                        =>* "if" "cond" <Then_Part> <Else_Part> ...
                        =>* "if" "cond" <Then_Part> "else" <Stmt> ...
                        =>* "if" "cond" <Then_Part> "else" ....
        - Reduce:
            5: <Else_Part> = .
                derived from:
                    3: <If_Stmt> = "if" "cond" . <Then_Part> <Else_Part>
                       by:
                    <Stmt>
                        =>* <If_Stmt> ...
                        =>* "if" "cond" <Then_Part> <Else_Part> ...
                        =>* "if" "cond" <Then_Part> "else" ....
                        =>* "if" "cond" "then" <Stmt> "else" ....
                        =>* "if" "cond" "then" <If_Stmt> "else" ....
                        =>* "if" "cond" "then" "if" "cond" <Then_Part>
                            <Else_Part> "else" ....
                        =>* "if" "cond" "then" "if" "cond" <Then_Part>
                            "else" ....

    The grammar is AMBIGUOUS

First derivation:
<Stmt>
   =>* <If_Stmt> ...
   =>* "if" "cond" <Then_Part> <Else_Part> ...
   =>* "if" "cond" <Then_Part> "else" ....
   =>* "if" "cond" "then" <Stmt> "else" ....
   =>* "if" "cond" "then" <If_Stmt> "else" ....
   =>* "if" "cond" "then" "if" "cond" <Then_Part> <Else_Part> "else" ....
   =>* "if" "cond" "then" "if" "cond" <Then_Part> "else" ....

Second derivation:
<Stmt>
   =>* <If_Stmt> ...
   =>* "if" "cond" <Then_Part> <Else_Part> ...
   =>* "if" "cond" <Then_Part> ...
   =>* "if" "cond" "then" <Stmt> ...
   =>* "if" "cond" "then" <If_Stmt> ...
   =>* "if" "cond" "then" "if" "cond" <Then_Part> <Else_Part> ...
   =>* "if" "cond" "then" "if" "cond" <Then_Part> "else" ....
```

```
Using the system disambiguating rules, priority is given to shift.
```

E.2 Une grammaire non LR(1)

On trouvera ci-dessous une grammaire non LR(1) et les messages de non-conformité émis par CSYNT dans le cas ou l'option `-path` est positionnée.

E.2.1 Grammaire

```
<Z>   = <A> <X> <T> ;
<Z>   = <B>         ;

<X>   =             ;

<T>   = t           ;

<B>   = a <E>       ;

<E>   = <C> <X> <T> ;

<A>   = a <D> <X>   ;

<C>   = <X> e       ;

<D>   = <X> e       ;
```

E.2.2 Listage des conflits avec l'option -path

```
SAMPLE PATH from state 1 to 9

"a" <X> "e"


    N O T   L A L R (1)
Reduce-Reduce conflict in state 9 on "t" between:
        - Reduce:
            8: <C> = <X> "e" .
                derived from:
                    6: <E> = . <C> <X> <T>
                       by:
                    <Z>
                        =>* "a" <E> ...
                        =>* "a" <C> <X> <T> ...
                        =>* "a" <C> "t" ....
                        =>* "a" <X> "e" "t" ....
        - Reduce:
            9: <D> = <X> "e" .
                derived from:
                    1: <Z> = . <A> <X> <T>
                       by:
                    <Z>
                        =>* <A> <X> <T> ...
                        =>* <A> "t" ....
                        =>* "a" <D> <X> "t" ....
                        =>* "a" <D> "t" ....
                        =>* "a" <X> "e" "t" ....
```

```
    The grammar is not LR (1)

First derivation:
<Z>
   =>* <A> <X> <T> ...
   =>* <A> "t" ....
   =>* "a" <D> <X> "t" ....
   =>* "a" <D> "t" ....
   =>* "a" <X> "e" "t" ....

Second derivation:
<Z>
   =>* "a" <E> ...
   =>* "a" <C> <X> <T> ...
   =>* "a" <C> "t" ....
   =>* "a" <X> "e" "t" ....

Using the system disambiguating rules, priority is given to reduction
number 8.
```

E.3 Spécifications d'une grammaire LALR(2)

E.3.1 Grammaire

* Grammaire (d'ecole) non LALR(1) mais LALR(2) du langage
* $L = \{e^n ca | e^n cb | e^n d | ae^n c | ae^n d\} \qquad n > 0$

```
*
<Z>        = <A, ca> c a  ;
<Z>        = <B, cb> c b  ;
<Z>        = <C> d        ;
<Z>        = a <B> c      ;
<Z>        = a <C> d      ;

<A, ca>    = <E> &1       ;

<B, cb>    = <E>          ;

<B>        = <E>          ;

<C>        = <E>          ;

<E>        = e            ;
<E>        = e <E>        ;

*
* &1 regarde si le contexte droit contient "c" et "ä
```

E.3.2 Actions

```
#include "sxunix.h"
#include "XNT_td.h" /* defines c_code to be 1 and a_code to be 2 */

int     XNT_parsact (entry, action_no)
    int         entry, action_no;
{
    switch (entry) {
    case OPEN:
    case CLOSE:
    case FINAL:
    case INIT:
        return;

    case PREDICATE:
        switch (action_no) {
        case 1:
            if (sxget_token (sxplocals.ptok_no)->lahead == c_code &&
                sxget_token (sxplocals.ptok_no + 1)->lahead == a_code)
                return TRUE;

            return FALSE;

        default:
            break;
        }

        break;

    default:
        break;
    }

    fputs ("The function \"XNT_parsact\" is out of date \
with respect to its specification.\n", sxstderr);
    abort ();
}
```

E.4 Un langage non-déterministe

E.4.1 Grammaire

```
* Langage non-deterministe
* L = {a^m b^2m a^n b^n | a^m b^m a^k a^n b^n}      m,k,n > 0
* Grammaire non LR (k) ni RLR ni LR (π)

<Z>    = @1 <S>        ;

<S>    = <A> <B'>      ;
<S>    = <B> <C> <B'>  ;

<A>    = a <A> b b     ;
<A>    = a b b         ;

<B>    = a <B> b       ;
```

```
<B>    = a b &1       ;

<B'>   = a <B'> b     ;
<B'>   = a b          ;

<C>    = @2 a &2 <C>  ;
<C>    = @2 a         ;

* init  : a_number = 1 et a_s = b_s = k = 0 ;
* @1    : Compte les premiers a et b dans a_s et b_s
*        Si a_s == b_s, compte les a et b de queue
*        et calcule k = |a| - |b|
* @2    : a_number++
*
* &1    : return a_s == b_s
* &2    : return a_number < k
```

E.4.2 Actions

```
#include "sxunix.h"
#include "NDL_td.h" /* defines a_code to be 1 and b_code 2 */

static int      a_s, b_s, a_number, k;

int     NDL_parsact (entry, action_no)
    int         entry, action_no;
{
    switch (entry) {
    case OPEN:
    case CLOSE:
    case FINAL:
        return;

    case INIT:
        a_number = 1;
        a_s = b_s = k = 0;
        return;

    case PREDICATE:
        switch (action_no) {
        case 1:
            return a_s == b_s;

        case 2:
            return a_number < k;
        }

        break;

    case ACTION:
        switch (action_no) {
        case 1: {
                register int i = sxplocals.atok_no, lahead;

#define GET_LAHEAD()  \
(lahead = sxget_token (i++)->lahead)
```

```
                while (GET_LAHEAD () == a_code)
                    a_s++;

                for (; lahead == b_code; GET_LAHEAD ())
                    b_s++;

                if (a_s == b_s) {
                    for (; lahead == a_code; GET_LAHEAD ())
                        k++;

                    for (; lahead == b_code; GET_LAHEAD ())
                        k--;
                }
            }

            return;

        case 2:
            a_number++;
            return;
        }
    }

    fputs ("The function \"NDL_act\" is out of date.\n", sxstderr);
    abort ();
}
```

Annexe F

Exemples de définitions lexicales

F.1 Définition lexicale de PASCAL

```
Tokens
    Comments     = -({SP|HT|EOL}+ |
                     "(" "*" {ANY} "*" ")"&True |
                     "{" {^"}"} "}") ;
    %IDENTIFIER = LETTER {LETTER | DIGIT} @UPPER_CASE ;
                   Context All But %IDENTIFIER, %NATUREL, %REEL ;
    %NATUREL     = {DIGIT}+ ;
                   Priority Reduce > Reduce ;
                   Unbounded Context All But %IDENTIFIER ;
    %REEL        = {DIGIT}+ ["." {DIGIT}+] [E ["+" | "-"] {DIGIT}+] ;
                   Context All But %IDENTIFIER ;
    %STRING      = -"'" {^"'" | -"'" "'"}+ -"'" ;
    "<>"         = -("<" ">" | "#") ;
    "["          = -("[" | "(" ".") ;
    "]"          = -("]" | "." ")") ;
    "@"          = -"@^" ;
```

F.2 Définition lexicale d'un langage à commentaires imbriqués

Cette section présente une alternative à la description des commentaires du langage OLGA présentée en figure 4.9.

F.2.1 Expressions régulières

Dans la description du langage principal (de niveau zéro) `llev.lecl`, on se contente de détecter le début d'un commentaire (ici "{") et on appelle l'action utilisateur `@0` :

```
Comments        = @0 "{" ;
```

Cette action appelle l'analyseur lexical du langage `hlev` qui se charge de la reconnaissance d'un niveau de commentaire. À un niveau quelconque, la reconnaissance d'une accolade ouvrante ("{") induit l'appel du niveau suivant par l'intermédiaire de l'action `@1` (essentiellement identique à l'action `@0`) alors que la reconnaissance d'une accolade fermante ("}"), vue par le niveau courant comme une fin de fichier, entraîne le retour au niveau précédent :

```
Comments        = "{" { ^"{}" | @1 "{" } ;
Eof             = "}" ;
```

F.2.2 Actions lexicales

```
#include "sxunix.h"

extern struct sxtables hlev_tables;

/* Structures used to store the local variables of a given scanner */
static struct sxsvar llev_svar, hlev_svar;


llev_sact (entry, action_no)
    int         entry;
    int         action_no;
{
    switch (entry) {
    case OPEN:
    case CLOSE:
        llev_svar = sxsvar;
        sxsvar = hlev_svar;
        (*(llev_svar.sxtables->analyzers.scanner)) (entry, &hlev_tables);
        hlev_svar = sxsvar;
        sxsvar = llev_svar;
        return;

    case INIT:
    case FINAL:
        return;

    case ACTION:
        switch (action_no) {
            struct sxsource_coord       source_coord;

        case 0 /* Call From Level 0 (llev scanner) */ :
            /* Swap of the scanner local variables sxsvar */
            llev_svar = sxsvar;
            sxsvar = hlev_svar;
            source_coord = sxsrcmngr.source_coord;
            (*(sxplocals.SXP_tables.scanit)) (); /* Quick Scanner Call */
            hlev_svar = sxsvar;
            sxsvar = llev_svar;

            if (sxsrcmngr.current_char != '}')
                sxput_error (source_coord, "%sComment not closed.",
                             sxsvar.sxtables->err_titles [2]);
```

```
            return;

        case 1 /* Call From Level Greater Than 0 (hlev scanner) */ :
            /* sxsvar is already correct */
            source_coord = sxsrcmngr.source_coord;
            (*(sxplocals.SXP_tables.scanit)) (); /* Quick Scanner Call */

            if (sxsrcmngr.current_char != '}')
                sxput_error (source_coord, "%sSub-Comment not closed.",
                             sxsvar.sxtables->err_titles [2]);

            return;
        }

    default:
        fputs ("The function \"llev_sact\" is out of date \
with respect to its specification.\n", sxstderr);
        abort ();
    }
}
```

On trouvera ci-dessous le résultat de l'exécution d'un petit texte source erroné.

```
            {1{2}{2}{2{3}2

text1.llev, line 1: column 9: Error:    Sub-Comment not closed.
            {1{2}{2}{2{3}2

text1.llev, line 1: column 1: Error:    Comment not closed.
```

F.3 Actions lexicales réalisant une inclusion

```
#include "sxunix.h"

VOID    your_scan_act (entry, act_no)
    int         entry, act_no;
{
    switch (entry) {
    case OPEN:
    case CLOSE:
    case INIT:
    case FINAL:
        sxincl_mngr (entry);
        return;

    case ACTION:
        switch (act_no) {
        /* The pathname of the include file is in token_string */
        int ste = sxstrsave (sxsvar.sxlv_s.token_string);
        /* it is saved (permanently) in the string_manager */
        char *path = sxstrget (ste);
        /* and get back. */
```

```
        case 1:
            if (sxpush_incl (path))
                /* The source text now comes from the include file */
                return;

            /* something failed (unable to open, recursive call, ...) */
            /* error message: */
            sxput_error (
                 sxsvar.sxlv.terminal_token.source_index
                      /* source coordinates of the include command */,
                 "%sUnable to process the include file \"%s\".",
                 sxsvar.sxtables->err_titles [2]
                      /* severity level: error */,
                 path /* include file name */
                 );
            /* however scanning of the current file is going on: */
            return;

        case 2:
            /* End of include processing */
            if (sxpop_incl ())
                return;

            /* something really wrong */
            /* error message */
            fputs ("Sorry, the include processing garbled, \
nevertheless hope to see you again soon.\n", sxstderr);
            abort () /* panic */ ;
        }

    default:
        fputs ("The function \"your_scan_act\" is out of date \
with respect to its specification.\n", sxstderr);
        abort ();
    }
}
```